UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

Tese

Uma Extensão à OpenMP para Suporte à Memória Transacional

André Desessards Jardim

Pelotas, 2021

André Desessards Jardim

Uma Extensão à OpenMP para Suporte à Memória Transacional

Tese apresentada ao Programa de Pós-Graduação em Computação do Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Doutor em Ciência da Computação.

Orientador: Prof. Dr. Gerson Geraldo H. Cavalheiro
Coorientador: Prof. Dr. André Rauber Du Bois

Pelotas, 2021

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

J37e Jardim, André Desessards

Uma extensão à OpenMP para suporte à memória transacional / André Desessards Jardim ; Gerson Geraldo H. Cavalheiro, orientador ; André Rauber Du Bois, coorientador. — Pelotas, 2021.

131 f. : il.

Tese (Doutorado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2021.

1. Programação multithread. 2. Memória transacional. 3. Memória transacional em software. 4. OpenMP. I. Cavalheiro, Gerson Geraldo H., orient. II. Bois, André Rauber Du, coorient. III. Título.

CDD : 005

Elaborada por Aline Herbstrith Batista CRB: 10/1737

André Desessards Jardim

Uma Extensão à OpenMP para Suporte à Memória Transacional

Tese aprovada, como requisito parcial, para obtenção do grau de Doutor em Ciência da Computação, Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas.

Data da Defesa: 21 de maio de 2021

Banca Examinadora:

Prof. Dr. Gerson Geraldo H. Cavalheiro (orientador)
Doutor em Computação pela Universidade Federal do Rio Grande do Sul.

Prof. Dr. André Rauber Dubois (coorientador)
Doutor em Computação pela University of Edinburgh.

Prof. Dr. Adenauer Correa Yamin
Doutor em Computação pela Universidade Federal do Rio Grande do Sul.

Prof. Dr. João Vicente Ferreira Lima
Doutor em Computação pela Universidade Federal do Rio Grande do Sul.

Prof. Dr. Vinícius Garcia Pinto
Doutor em Computação pela Universidade Federal do Rio Grande do Sul.

Dedico esse trabalho aos meus pais Pedro Osório da Conceição Jardim (in memorian) e Suzana Maria Desessards Jardim, exemplos de caráter e dignidade, com todo o meu amor e gratidão.

AGRADECIMENTOS

Agradeço primeiramente ao professor Adenauer Correa Yamin, por me colocar em contato com o professor Gerson Geraldo H. Cavalheiro, me permitindo assim entrar no curso de Doutorado em Computação da UFPel.

Aos colegas Henry S. Pereira, Kevin O. de Oliveira, pela ajuda inestimável nas implementações dos códigos.

Ao colega Edevaldo dos Santos, pela ajuda mútua nesses nossos anos de Doutorado e pela amizade durante esse tempo.

Ao coorientador André Rauber Dubois, pelas anotações e comentários no texto, imprescindíveis para as correções e o aprimoramento do mesmo.

À minha família, principalmente à minha mãe, sempre me incentivando em cada etapa da minha vida.

À minha namorada Val, simplesmente por ser quem ela é e estar sempre ao meu lado.

E finalmente ao meu orientador Gerson Geraldo H. Cavalheiro, por aceitar ser o meu orientador, pela amizade, pelo incentivo, pela orientação, pela imensa paciência e pela grande ajuda durante todo o trabalho, principalmente na reta final. Obrigado por tudo, Gerson!

Pus-me a escrever o que sei dizer, adaptando o meu assunto às minhas forças.
— Montaigne

RESUMO

JARDIM, André Desessards. **Uma Extensão à OpenMP para Suporte à Memória Transacional**. Orientador: Gerson Geraldo H. Cavalheiro. 2021. 131 f. Tese (Doutorado em Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2021.

Um dos aspectos mais complexos no desenvolvimento de programas em ambientes com memória compartilhada é a sincronização de atividades concorrentes no acesso a dados compartilhados. Memória Transacional foi proposta como um mecanismo que abstrai algumas das complexidades associadas ao acesso concorrente a dados compartilhados, enquanto promove o desenvolvimento de programas mais legíveis pela oferta de uma interface de programação de mais alto nível. 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 na sincronização de fluxos de execução. O objetivo do trabalho é o de estender o estado da arte em interfaces para programação concorrente multithread pela introdução de recursos para manipulação de Memória Transacional em ferramentas de programação consolidadas. O objetivo foi alcançado pela caracterização de uma extensão à OpenMP, permitindo a manipulação de dados de forma transacional. Diferente de outras abordagens similares encontradas na bibliografia, a proposta apresentada se destaca por realizar a sincronização sobre o dado e não nos fluxos de execução, como previsto no próprio modelo de Memória Transacional. Na tese, é apresentada a especificação da extensão proposta, sua prototipação e sua validação qualitativa e quantitativa. A validação qualitativa se deu pela comparação, considerando métricas obtidas por análise de código, entre os códigos obtidos em implementações de diferentes aplicações em programas empregando a extensão proposta com outras soluções. Esta análise indicou que a solução apresentada atendeu os requisitos de abstração desejados. O protótipo, construído para verificar a viabilidade de implementação da interface, foi também avaliado em termos de desempenho, na análise quantitativa. O protótipo foi construído na forma de uma linguagem intermediária, permitindo instanciar o programa sobre diferentes ferramentas de suporte à Memória Transacional em software. As análises de desempenho consideraram um variado conjunto de casos de estudo, e as análises dos resultados permitiram atestar a viabilidade de implementação da interface proposta.

Palavras-chave: Programação Multithread. Memória Transacional. Memória Transacional em Software. OpenMP.

ABSTRACT

JARDIM, André Desessards. **An OpenMP Extension to Support Transactional Memory**. Advisor: Gerson Geraldo H. Cavalheiro. 2021. 131 f. Thesis (Doctorate in Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2021.

One of the most complex aspects of developing programs in environments with shared memory is the synchronization of concurrent activities accessing shared data. Transactional Memory was proposed as a mechanism that abstracts some of the complexities associated with concurrent access to shared data while promoting the development of more readable programs by offering a higher-level programming interface. Although the modern tools for multithreaded programming offer resources for efficient exploitation of the hardware, the supports for synchronization to shared data still reflect models based on the synchronization of execution flows. The aim of the work is to extend the state of the art in interfaces for multithreaded concurrent programming by introducing features for handling Transactional Memory in consolidated programming tools. The objective was achieved by the characterization of an extension to OpenMP, allowing the manipulation of data in a transactional way. Unlike other similar approaches found in the bibliography, the proposal presented stands out for performing synchronization on the data and not on the execution flows, as predicted in the Transactional Memory model itself. In the thesis, the specification of the proposed extension, its prototyping, and its qualitative and quantitative validation are presented. Qualitative validation took place by comparison, considering metrics obtained by code analysis, between the codes obtained in implementations of different applications in programs using the proposed extension with other solutions. This analysis indicated that the solution presented met the desired abstraction requirements. The prototype, built to verify the feasibility of implementing the interface, was also evaluated in terms of performance, in the quantitative analysis. The prototype was built in the form of an intermediate language, allowing to instantiate the program on different tools to support Transactional Memory in software. The performance analyzes considered a varied set of case studies, and the analysis of the results allowed to attest the feasibility of implementing the proposed interface.

Keywords: Multithreaded Programming. Transactional Memory. Software Transactional Memory. OpenMP.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

OpenMP	Open Multi-Processing
TBB	Thread Building Blocks
TM	Transactional Memory
STM	Software Transactional Memory
HTM	Hardware Transactional Memory
API	Application Programming Interface
GCC	GNU Compiler Collection
CRCW	Concurrent Read Concurrent Write
EREW	Exclusive Read Exclusive Write
ERCW	Exclusive Read Concurrent Write
CRCW	Concurrent Read Concurrent Write
OpenTM	Open Transactional Memory
E/S	Entrada/Saída
OTM	Open Transactional Memory
GC	Gradientes Conjugados
CDT	Conflict Detection Thread
BUSTM	Benchmark for Unstructured-Mesh Software Transactional Memory
GQM	Goal Question Metric
NLC	Número de Linhas do Código
QRU	Quantidade de Recursos Utilizados
QIDP	Quantidade de Invocações a Diretivas de Paralelização
R/W	Read Write
R/W/RW	Read/Write/Read Write
RAM	Read Only Memory

SUMÁRIO

1 INTRODUÇÃO

A exploração de concorrência por um programa, em um modelo de programação multithreaded, se dá pela execução simultânea de diversos fluxos de instruções sobre os recursos de uma mesma arquitetura. Neste modelo de programa os fluxos de execução de instruções, denominados de threads, além de compartilhar no tempo, o acesso ao processador, e também os demais recursos da arquitetura, são concebidos de forma a cooperar entre si por meio de variáveis armazenadas em um espaço de endereçamento compartilhado (CAVALHEIRO; DU BOIS, 2014). Nas gerações atuais de ferramentas de programação multithreaded, em particular naquelas não restritas ao meio acadêmico ou de pesquisa, cabe ao próprio programador introduzir mecanismos sincronização entre os threads que garantam integridade no acesso aos dados compartilhados.

A programação concorrente, inicialmente, teve uso voltado ao desenvolvimento de sistemas operacionais confiáveis, foi rapidamente compreendida como aplicável "a qualquer forma de computação paralela", nos anos 1960 (HANSEN; DIJKSTRA; HOARE, 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 (SKILLICORN; 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.

Segundo (WILLIAMS, 2012), as principais motivações para exploração de programação concorrente estão relacionadas: (*i*) à separação de interesses, onde agrupando pedaços de código relacionados entre si e mantendo pedaços independentes de código separados, permite que os programas sejam mais fáceis de entender e testar; e (*ii*) à necessidade de aumento de desempenho na execução de programas. No entanto, um terceiro aspecto, que diz respeito à onipresença de arquiteturas multiprocessadas nos sistemas computacionais, pode ser considerado. Nesta realidade de alta disponibilidade de arquiteturas paralelas, a exploração da programação con-

corrente passa a ser uma necessidade a toda e qualquer aplicação, sob pena, na sua falta, de subtilização dos recursos disponíveis.

No presente trabalho, os três aspectos de interesse citados para o uso da programação concorrente são considerados. O foco da pesquisa é buscar alternativas que permitam aumentar a eficiência no uso dos recursos paralelos do hardware disponível ao mesmo tempo que garantam níveis de produtividade no ciclo de desenvolvimento de código.

As ferramentas de programação utilizadas nos dias de hoje oferecem os mesmos mecanismos de sincronização concebidos há três décadas, época na qual os problemas de sincronização se apresentavam em uma escala menor. Nas primeiras aplicações da concorrência, haviam menos *cores* disponíveis ao processamento e a maioria das aplicações se dava no desenvolvimento de software básico. À medida que os programas multithread aumentam em tamanho e complexidade, abstrações mais avançadas se fizeram 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).

Na programação concorrente, a sincronização 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). Como threads concorrentes se comunicam por acessos em escrita e/ou leitura sobre variáveis compartilhadas, acessos aos dados compartilhados se dão em seções críticas de código, as quais necessitam ser explicitamente sincronizadas a fim evitar interferências entre os threads na manipulação dos dados e garantir execução em regime de exclusão mútua (RIGO; CENTODUCATTE; BALDASSIN, 2007).

As arquiteturas multicore exigem programação concorrente, especialmente programação multithreaded. Escrever programas concorrentes é desafiador: programas multithread frequentemente sofrem de bugs de concorrência, como impasses (*deadlocks*), condições de corrida, violações de atomicidade e violações de ordem (ADL-TABATABAI et al., 2006), que são difíceis de expor (OWENS, 2010), detectar (BAEK et al., 2007), (BAEZA-YATES; RIBEIRO-NETO, 1999), (Bienia et al., 2008), depurar (Bienia et al., 2008), e reparar (BUTENHOF, 1997), (CASCAVAL et al., 2008) devido à sua natureza não determinística. Portanto, evitar erros de concorrência em tempo de execução é uma abordagem complementar atrativa para obter programas mais robustos.

Situando a discussão no contexto da programação em multiprocessadores, é es-

perada a formação de programadores no uso de ferramentas para programação multithreaded. Esta formação, atualmente remete ao domínio de recursos clássicos de programação, como mutex e semáforos, cuja proposta data dos anos 1960. Ferramentas de mercado para programação mulithreaded modernas, como C++, Java e OpenMP oferecem tais recursos de programação, mesmo que abstrações mais elaboradas para sincronização a dados compartilhados estejam disponíveis. Mesmo TBB (*Thread Building Blocks*), representante de uma nova geração de ferramentas de programação multithreaded, 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 Memória Transacional (TM, *Transactional Memory*)[1].

O modelo de TM surgiu como alternativa promissora aos mecanismos de sincronização baseados em exclusão mútua. Podendo ser implementada em hardware ou software, Memória Transacional oferece uma alternativa com mais alto nível de abstração aos mecanismos clássicos de sincronização, fornecendo um novo construtor de controle de sincronização que evita problemas comuns de bloqueios e simplifica significativamente o esforço de programação para produzir o software correto (HARRIS; LARUS; RAJWAR, 2010).

Um forte argumento para o uso de TM se dá a partir das vantagens de sua utilização em relação ao *mutex*: escalabilidade, composibilidade, robustez e redução da contenção (HARRIS et al., 2005a; HARRIS; LARUS; RAJWAR, 2010; WAMHOFF et al., 2010). Outro importante argumento é sua capacidade de expressão. No trabalho de (KESTOR et al., 2011), é estudada a definição de um *benchmark* para Memórias Transacionais, avaliando critérios de implementação e execução para demonstrar o ganho quantificado da utilização de transações. Já o trabalho de (VOLOS et al., 2012a) apresenta uma aplicação de TM para lidar com *bugs* de concorrência, demonstrando que mesmo para problemas complexos é possível alcançar a expressividade dos exemplos escolhidos.

Memória Transacional tem, como um de seus objetivos principais, facilitar a atualização atômica na memória de múltiplos dados independentes, evitando problemas relacionados a *locks* (impasses, inversão de prioridades e *convoying*). A utilização

[1] Neste texto, a abreviatura TM é utilizada como referência ao termo *Memória Transacional*.

de TM fornece ao sistema um modelo de programação paralela mais abstrato, com capacidade composicional e com alto desempenho para sistemas de multiprocessador, tornando-se um modelo de programação promissor (SRIVATSA; KUMAR, 2012). Memórias Transacionais, embora existam desde os anos 90, não foram efetivamente absorvidas na nova geração de ferramentas para programação multithreaded.

APIs (*Application Programming Interface*) de programação multithreaded modernas, como OpenMP, C++ e TBB, no entanto, não incluem um modelo de programação com TM de maneira nativa. Os trabalhos de (BAEK et al., 2007; MILOVANOVIĆ et al., 2008; WONG et al., 2010, 2014) apresentam esforços de realizar esta inclusão por meio de extensões. Em comum, tais propostas apresentam uma interface de manipulação de transações que estendem a oferecida por OpenMP pela introdução de uma nova diretiva específica para tratar transações, e constituem um ótimo ponto de partida para o estudo de transações dentro do OpenMP. Tal estudo é o alvo deste trabalho.

Estes trabalhos são propostas antigas que não levam em consideração abstrações modernas do Openmp, a organização da memória, recursão e inclusão e exclusão de variáveis do escopo da transação, além de possuírem projetos geralmente fixos em uma implementação de OpenMP. Essas propostas também levam em consideração bibliotecas de Memória Transacional de Software (STM) fixas para tratar transações, e não existe uma configuração de algoritmo de TM que funcione bem para todos os casos.

1.1 Hipótese e Questões de Pesquisa

Considerando as motivações já discutidas e os trabalhos relacionados estudados, a **Hipótese de Pesquisa** construída para esta tese foi: estender a interface de programação de uma ferramenta de programação multithreaded com recursos de sincronização baseados em Memória Transacional permite a construção de programas com maior nível de abstração e com desempenho aceitável considerando o contexto de execução suportando Memória Transacional. O caso de estudo nesta tese é a ferramenta de programação multithread OpenMP.

Para avaliar a hipótese apresentada, três questões de pesquisa foram elencadas para resumir o esforço a ser desenvolvido para validar, ou refutar, a hipótese apresentada. Os escopos destas questões se intercalam, oferecendo um esforço de pesquisa com várias oportunidades de avanço.

QP1: Como deve ser composta uma proposta para estender o estado da arte da especificação de interfaces de programação de ferramentas de programação multithreaded?

QP2: Como estender a API de OpenMP para suportar Memória Transacional?

QP3: Qual o impacto do uso de Memórias Transacionais para simplificação do código concorrente e no seu desempenho?

As ferramentas atuais oferecem o recurso de TM na forma de bibliotecas de serviço, não estando, portanto, vinculadas às ferramentas de programação multithreaded. Assim, embora possam ser utilizadas, as ferramentas TM existentes não refletem o modelo de programação adotado pelas ferramentas multithread. A hipótese de pesquisa considerou ser possível o desenvolvimento de uma proposta que estenda o estado da arte da especificação de interfaces de programação de ferramentas de programação multithreaded. Neste sentido, buscou contemplar o uso de TM em programas OpenMP, uma interface de programação multithreaded popular na comunidade de programação concorrente. Isso se deu pela introdução de um mecanismo de suporte a Memórias Transacionais integrado à interface de programação desta ferramenta. A implementação de tal proposta visou garantir que a ferramenta original não tivesse suas características de programação alteradas pela introdução dos novos recursos. Considerando o estudo de caso em OpenMP, a hipótese foi avaliada analisando o impacto, em termos de legibilidade de código e de desempenho, da extensão proposta na construção de programas em OpenMP padrão e em OpenMP estendido.

A diferença na abordagem desta tese em relação aos trabalhos estudados está em considerar o impacto do uso de Memória Transacional com alguns construtores de paralelização destas ferramentas. A contribuição do trabalho 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 multithreaded. Como resultados, destaca-se o desenvolvimento mais robusto de código, e com possibilidade de composabilidade. Artigos para eventos na área também foram contemplados.

Considerando a questão do compartilhamento de dados entre tarefas, observa-se uma lacuna em OpenMP que pode ser coberta pela adoção de suporte à Memória 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.

Esta tese propõe uma interface para Memória Transacional que estende a capacidade de expressão da linguagem base, no caso, OpenMP. Isso se dá pela incorporação de um novo recurso na sua interface de programação. Neste ponto, o ganho obtido é associar a semântica das operações sobre Memória Transacional com a oferecida pelas outras operações já disponíveis em OpenMP. A presente proposta, apesar de semelhante às apresentadas nos trabalhos estudados, diferencia-se pela

forma como os identificadores que devem ser transacionados são definidos: a solução proposta integra o modelo de Memória Transacional ao modelo de programação de OpenMP. Desta forma, a extensão proposta oferece uma nova funcionalidade à própria linguagem OpenMP, sem modificar os construtores nela já existentes e ampliando o seu poder de expressão. Além disso, entre todas as abordagens estudadas, nenhuma possui recurso para especificar o modo de acesso (leitura ou escrita) para os identificadores da transação. A proposta apresentada inclui a possibilidade desta especificação, o que pode ser explorado pela ferramenta responsável pela manutenção da Memória Transacional propriamente dita.

1.2 Objetivos

O objetivo do trabalho é o de estender o estado da arte em interfaces para programação concorrente multithreaded pela introdução de recursos para manipulação de Memórias Transacionais em ferramentas de programação consolidadas. O caso de estudo é a ferramenta OpenMP. O diferencial buscado é o de garantir que OpenMP não tenha suas características de programação alteradas pela introdução dos novos recursos.

A introdução de mecanismos de Memórias Transacionais em ferramentas de programação multithreaded é estudada no que se refere às questões de interface de programação. Entende-se que sua adoção efetiva possa se dar quando seu uso for consonante com as decisões de projeto da interface de programação nativa. Esta tese procura estender o estado da arte da especificação de interfaces de programação de ferramentas de programação multithreaded pela introdução de um mecanismo de suporte à Memórias Transacionais. O diferencial deste objetivo, em relação a trabalhos já desenvolvidos, é observar que a extensão propostas seja integrada a interface de programação da ferramenta multithread e não um apêndice à esta ferramenta, representado por uma biblioteca de serviços.

Este trabalho propõe uma abordagem, no intuito de fornecer OpenMP como uma ferramenta para implementações, explorando o modelo de Memória Transacional. Neste contexto, respeitando a premissa de não modificar as características da programação OpenMP, é analisado o impacto do uso de um novo modo de parametrização às tarefas, indicando a presença de dados em Memória Transacional. Também pretende-se garantir a compatibilidade dos dados manipulados no programa com os serviços sobre a Memória Transacional. O foco do trabalho é, portanto, oferecer a interface OpenMP para programadores de TM. Assim, o desafio que se soma é o de fornecer tal benefício de maneira eficiente no uso do resultado obtido, de tal forma que não impeça ganhos potenciais do paralelismo explorado.

1.3 Metodologia

O processo metodológico de desenvolvimento inicia por um estudo bibliográfico onde são identificadas lacunas em trabalhos anteriores que justifiquem a construção da proposta apresentada. A etapa imediatamente posterior é a especificação da extensão propriamente dita, considerando a interface de OpenMP. As etapas metodológicas subsequentes envolvem a validação da programabilidade da extensão, sua prototipação e, então, validação do protótipo.

A validação da programabilidade é realizada comparando métricas obtidas a partir da análise do código obtido da implementação de programas escritos em OpenMP suportando a interface proposta, com o código dos mesmos programas implementados com: OpenMP puro, OpenMP estendido por interfaces de programação com propostas similares encontradas na literatura e por bibliotecas de oferecendo Memória Transacional em Software. Para realização deste experimento, o conjunto de programas selecionados pertencem ao benchmark Cowichan (WILSON; IRVIN, 1995). Este benchmark caracteriza-se por ter sido desenvolvido especialmente para avaliar o poder de expressão de linguagens paralelas, sendo, portanto, adequado à esta etapa de validação.

A prototipação da solução se deu pela incorporação, em OpenMP, do uso de bibliotecas para manipulação de Memória Transacional. Para permitir a compatibilização do protótipo com diferentes suporte a TM, foi concebida uma linguagem intermediária, *Vanilla*-TM, para representar os serviços das diferentes bibliotecas. Nesta etapa do trabalho, foram utilizados os recursos para programação com Memória Transacional oferecido por GCC-TM (HONORIO, 2018) e por TinySTM (FELBER; FETZER; RIEGEL, 2008). A validação do protótipo se deu pela execução dos programas Cowichan implementados e coleta de tempos.

A análise de desempenho foi finalmente conduzida, buscando analisar o comportamento do protótipo frente a dois casos distintos. O primeiro quando a solução proposta é utilizada para implementar um benchmark desenvolvido explicitamente para OpenMP. O segundo, na implementação de um benchmark desenvolvido explicitamente para avaliar desempenho de ferramentas de programação com Memória Transacional. Também em relação a análise de desempenho, de forma mais específica, análise de comportamento de execução, é feito um estudo considerando o uso da interface proposta em algoritmos recursivos.

De forma resumida, a metodologia apresentada envolve conhecer as soluções de TM para ferramentas OpenMP, estendendo a API de ferramentas de programação multithreaded, para suportar Memória Transacional, tendo como caso de estudo, OpenMP, a fim de validar uma proposta buscando uma alternativa que não altere as características de programação do OpenMP. Oferecendo assim, OpenMP para que

programadores TM explorem uma nova interface de programação.

1.4 Contribuições Científicas

As contribuições científicas deste trabalho dão-se no contexto da computação paralela. Mais especificamente em interfaces para programação multithreaded e aplicação do conceito de Memória Transacional. De forma explícita, as contribuições são:

- avanço no estado da arte das interfaces para programação multithreaded com a inclusão de suporte ao modelo de Memória Transacional em Software;
- avanço na compreensão dos limites de desempenho na adoção de TM em interfaces de programação multithreaded;
- Implementação de programas do benchmark Cowichan com OpenMP e suporte de TM;
- apresentação de uma representação intermediária para construção de programas OpenMP com diferentes suportes de TM em Software;
- análise de um estudo de caso de aplicação de Memória Transacional em um algoritmo recursivo implementado em OpenMP.

Com estas contribuições, espera-se contribuir com a expansão das fronteiras de pesquisa em interfaces de programação paralela, oferecendo uma alternativa para adoção de TM em OpenMP.

1.5 Organização do Texto

O Capítulo 2 apresenta uma fundamentação teórica referente aos tópicos abordados na tese, apresentando os conceitos estudados. O capítulo também aponta aspectos considerados das ferramentas exploradas na realização do trabalho.

O Capítulo 3 apresenta o estado da arte, abordando os trabalhos na literatura com propostas semelhantes, de integração de TM com OpenMP. Também relaciona trabalhos que discutem a integração de TM com ferramentas de programação multithreaded, outras que não OpenMP.

No Capítulo 4 é apresentada a interface proposta na Tese e um modelo de memória de OpenMP com transações é contemplado, com uma análise do código obtido. Também é apresentada a prototipação da interface proposta, na forma de uma linguagem intermediária.

O Capítulo 5 apresenta a prova de conceito, com a prototipação, implementação e experimentação da interface proposta, assim como a validação da interface, a partir da análise dos resultados obtidos.

E por fim, no Capítulo 6, são apresentadas as conclusões e considerações finais do trabalho, assim como possibilidades para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos dos referenciais teóricos explorados neste trabalho: programação concorrente, Memória Transacional e aspectos relacionados à utilização de mecanismos clássicos de sincronização baseados em regiões críticas e exclusão mútua. O capítulo também aponta aspectos considerados das ferramentas exploradas na realização do trabalho, o benchmark Cowichan e a ferramenta de programação multithread OpenMP. Por fim, o capítulo apresenta as ferramentas de programação suportando Memória Transacional, utilizadas como suporte de implementação: GCC-TM e TinySTM.

2.1 Mecanismos Clássicos para Exclusão Mútua

A popularização dos microprocessadores multicore implicou no aumento considerável da demanda na área do processamento paralelo. Neste contexto, durante a evolução dos threads que compõem um programa, existe a necessidade de colaboração, via compartilhamento de dados em memória. Mecanismos de sincronização e coordenação são, portanto, essenciais.

Os modelos tradicionais de programação multithreaded geralmente oferecem um conjunto de primitivas de baixo nível, como mutexes, a fim de garantir a exclusão mútua, para que os threads não sobrescrevam informações uns dos outros, tornando o resultado inconsistente. A utilização de um ou mais mutexes permite sincronizar diferentes fluxos de execução, coordenando-os no acesso a dados compartilhados. Porém, a programação com controle de sincronização por mutex é complexa e contra intuitiva, podendo, inadvertidamente, o(s) programador(es) introduzir(em) erros no programa pela sua má utilização, notadamente, condições de corrida ou situações de postergação indefinida (SILBERSCHATZ; GALVIN; GAGNE, 2010).

O compartilhamento de dados não sincronizado entre threads pode levar a uma situação indesejável, a **condição de corrida**. Este caso é caracterizado pela existência de dois ou mais threads acessando um dado compartilhado ao mesmo tempo e na qual pelo menos um dos threads realiza uma operação de escrita. Para evi-

tar problemas de compartilhamento, **seções críticas** – trechos de código em que os threads estão evoluindo computações sobre dados compartilhados com outros threads – devem ser detectadas e protegidas de maneira a evitar as condições de corrida. Este conceito é conhecido como **exclusão mútua**. Para evitar que o dado seja corrompido, não podem haver dois ou mais threads na mesma seção crítica, assim como nenhum thread bloqueado fora de sua seção crítica pode bloquear outros threads, e nenhum thread deve esperar um tempo demasiadamente longo para entrar em sua seção crítica (SILBERSCHATZ; GALVIN; GAGNE, 2010).

2.2 A Programação Concorrente

A programação concorrente, ou paralela, surgiu da necessidade de explorar eficientemente recursos de hardware oferecidos pela arquitetura em uso, tendo sido proposta e elaborada, em seus estágios iniciais, no contexto do desenvolvimento de sistemas operacionais (TANENBAUM, 2010), (SILBERSCHATZ; GALVIN; GAGNE, 2010). Novas aplicações deste paradigma de programação passaram a ser dadas no desenvolvimento de aplicações para o processamento de alto desempenho quando do surgimento de arquiteturas paralelas, mais particularmente, no contexto deste trabalho, de arquiteturas multiprocessadas. Neste caso, um dos objetivos a ser alcançado pelo uso das ferramentas de programação concorrente, as chamadas ferramentas de programação multithreaded, passou a ser explorar o paralelismo disponível na arquitetura para obtenção de melhores índices de desempenho dos programas (WILLIAMS, 2012).

A portabilidade é um requisito importante, pois o tempo de vida do hardware é relativamente curto. Os sistemas de software não devem depender, portanto, de uma arquitetura em particular, ou seja, devem executar de forma eficiente em máquinas com diferentes arquiteturas (SEBESTA, 2009). A portabilidade, assim, não deve ser entendida apenas em termos de portabilidade do código em si, mas, no mundo do processamento paralelo, também no contexto da portabilidade de desempenho, ou seja, na escalabilidade do desempenho em função do incremento do número disponível de CPUs de processamento.

A programação concorrente, nas suas mais diversas formas, tem sido cada vez mais utilizada nos meios acadêmico, científico e de produção. Este aumento de interesse, apesar da programação concorrente não ser uma técnica de programação realmente nova, se deu devido à maior acessibilidade à recursos computacionais paralelos – as arquiteturas multiprocessadoras e os agregados de computadores (BRESHEARS, 2009).

Existem ao menos duas razões para projetar sistemas de software concorrentes, atendendo a necessidade da produção em larga escala de aplicações. A primeira

é obter melhor desempenho na execução dos programas, podendo este desempenho ser medido de diferentes formas, sendo o *tempo de execução* o mais usual. Hardware concorrente fornece uma maneira efetiva de aumentar a velocidade de execução de um programa, desde que os programas sejam projetados para usar de forma eficiente o paralelismo disponibilizado pelo hardware. A segunda razão é que a concorrência oferece seu próprio conjunto de abstrações para modelagem dos programas de aplicação, podendo-a tornar um modelo de programação mais adaptado para determinado conjunto de problemas, naturalmente concorrente, que o paradigma de programação sequencial tradicional (SEBESTA, 2009).

O desempenho atingido pelos computadores durante os últimos anos, em função da popularização do uso da tecnologia multicore, cresceu exponencialmente. Para fazer uso do potencial do hardware atual, o software deve ser paralelizado. É preciso explorar a concorrência existente nas aplicações, migrando, de um modelo de desenvolvimento de programas sequenciais, para um modelo de programação que tire proveito do paralelismo presente no hardware. Neste novo contexto, a programação concorrente é bem mais difícil do que a programação sequencial, principalmente no que diz respeito ao desafio de se produzir um código correto, seguro e robusto. Um dos grandes desafios está no desenvolvimento de novos mecanismos para programação concorrente que, além de serem tangíveis para a maioria dos programadores, também possam tirar o máximo desempenho possível dos processadores multicore (WILLIAMS, 2012).

Neste trabalho considera-se a distinção dos termos programação concorrente e programação paralela. Quando um programa é descrito de forma *concorrente*, o programador supõe que diferentes partes que compõem o programa podem progredir ao mesmo tempo, disputando, eventualmente dados compartilhados. O número de atividades concorrentes é definido em função das características da aplicação. Na programação *paralela*, diferentes partes do programa evoluem, de fato, simultaneamente. Neste caso, o número de atividades concorrentes do programa é definido em termos da capacidade de processamento efetivamente paralelo do hardware. O conceito *progredir*, empregado na definição de concorrência é chave nesta distinção: *progredir* indica que as diferentes partes do programa não necessitam estar, de fato, executando, ou *evoluindo* ao mesmo tempo. O termo concorrente ainda destaca que as diferentes partes do programa em execução competem por recursos. Sob a ótica de um programador de aplicação, estes recursos, objetos de competição, são os dados manipulados no programa, os quais devem ser acessados de forma coordenada entre as atividades em execução (NICHOLS; BUTTLAR; FARRELL, 1996), (BRESHEARS, 2009). A competição por recursos de hardware, como CPU e memória, também existe. No entanto, esta forma de competição é tratada em nível de sistema operacional, com maior ou menor apoio do próprio hardware, conforme a implementação. A

competição por recursos de hardware pode ser, então, abstraída pelo programador no desenvolvimento de seu software, não sendo objeto de consideração no presente trabalho.

Nos dias atuais, considerando a onipresença de arquiteturas multiprocessadas promovida pela popularização de arquiteturas multicore, o espectro da programação concorrente se abre para uma gama maior de aplicações. A importância do uso de ferramentas de programação multithreaded, cresce a medida em que as arquiteturas multicore se consolidam. Obter alto desempenho, ou, pelo menos, eficiência na execução, continua sendo um dos objetivos a ser alcançado. No entanto, outras características são desejáveis aos programas (SEBESTA, 2009), como robustez, capacidade de manutenção, segurança e tamanho. Considerando a necessidade de desenvolvimento de programas concorrentes, neste trabalho, o olhar é dado às questões relacionadas aos dois primeiros itens: robustez e capacidade de manutenção.

2.2.1 Premissas da Programação Concorrente

A concorrência foi introduzida nos sistemas de computação como um meio para aumentar o desempenho do sistema minimizando a ociosidade da(s) CPU(s). Isto imediatamente impulsionou a programação concorrente e o desenvolvimento de mecanismos e sistemas operacionais que suportassem a concorrência e o paralelismo de operações. Existem dois modelos básicos para construção de programas concorrentes. O primeiro, **concorrência de tarefas**, onde o algoritmo é decomposto em tarefas independentes, podendo cada uma ser executada concorrentemente. O segundo, **concorrência de dados**, onde um espaço de dados é decomposto e atribuído à computações independentes; cálculos são executados em threads separados que são coordenados com sincronização explícita.

O controle de concorrência visa gerenciar o acesso a recursos compartilhados, representados, no programa, por dados. O objetivo deste controle é permitir o acesso, sem conflitos, aos recursos compartilhados. Tal controle é importante, pois o acesso concorrente a dados e recursos partilhados pode criar uma situação de inconsistência. Portanto, para que uma rotina ou programa seja consistente são necessários mecanismos que assegurem a execução ordenada e correta dos threads cooperantes.

Em sistemas multithread, frequentemente, os threads que compõem um programa precisam trocar informações entre si ou solicitam a utilização de um mesmo recurso simultaneamente, como arquivos, registros, dispositivos de E/S e memória. O compartilhamento de recursos entre vários processos pode causar situações indesejáveis e, dependendo do caso, gerar o comprometimento da aplicação. É necessário gerenciar e sincronizar processos concorrentes, estabelecendo uma forma estruturada de se fazer esta comunicação, com o objetivo de manter o bom funcio-

namento do sistema.

2.3 Problemas com Mutex

No que diz respeito às questões de desempenho, estratégias globais de bloqueio, resultantes do uso de mutexes granularidade grossa, onde todos os dados do programa são protegidos usando um ou poucos mutexes, levam à serialização desnecessária da execução do programa e, consequentemente, menor desempenho pela perda do potencial de exploração do paralelismo do hardware. O bloqueio de granularidade grossa resulta em uma 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 impasses. O bloqueio de granularidade fina, por outro lado, além de resultar em uma associação complexa entre dados e mecanismos de sincronização que protege o acesso a esses dados, implica em um grande número de sincronizações, sempre custosas de serem efetivadas.

Impasses representam também um erro de programação. Neste caso, a falha de execução observada consiste em um thread não poder prosseguir por estar aguardando alguma condição sincronização que jamais será atendida.

Um sistema de software pode ser formado inteiramente somente por componentes, pois estes se interligam através de suas interfaces. Este processo de comunicação entre componentes é denominado composição. Outro problema com a utilização de mutexes diz respeito à composição de software. Por exemplo, supondo que se tenha uma tabela hash, implementada usando mutexes, ou seja, todos os métodos adquirem mutexes no início e liberam mutexes no final. Se for necessário transferir um elemento de uma tabela hash para outra, a implementação já não funciona, pois tirar um elemento de uma tabela e colocar em outra abre um espaço em que mutexes não estão adquiridos, alterando o estado do programa. Os esforços de desenvolvimento de software atual requerem reaproveitamento de código e também, não raro, implicam desenvolvimento de código por equipes diferentes. Falta de documentação adequada e comunicação entre as equipes pode levar a programas mal escritos

Existem também dois aspectos de desempenho a serem considerados no uso de mutex: o sobrecusto de execução devido a trocas de contexto entre threads, e a serialização na execução de seções críticas (PANKRATIUS; ADL-TABATABAI, 2014). Operacionalmente o mecanismo de mutex cumpre sua função. No entanto, a estrutura de um mutex não permite associar o mecanismo de sincronização a um dado específico. O desempenho, a robustez e a manutenibilidade do código obtido depende do algoritmo implementado e da atenção do programador no seu desenvolvimento.

Sob a ótica dos processadores multicore, mutexes possuem como desvantagem a limitação de paralelismo. Somente um thread, o que detiver a posse do mutex, pode evoluir sobre uma seção crítica em um determinado instante de tempo, introduzindo um ponto de contenção e serialização da execução. Qualquer tentativa de outro thread evoluir na seção crítica enquanto a mesma estiver ocupada implica em bloqueio do thread solicitante, aguardando que ele próprio obtenha o mutex correspondente.

A falta de manutenção correta dessas associações ao longo do programa leva a erros de concorrência, como condições de corrida e postergações indefinidas, diminuindo a robustez do programa. A desatenção, e a eventual falta de capacitação dos programadores (AHMED; BAGHERZADEH, 2018), também resulta em fragilidade do código produzido. 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 custos diferentes para as primitivas de sincronização.

Além de arriscar o surgimento de novos bugs, os mutexes também não suportam muito bem a programação na qual as equipes de desenvolvimento distribuído criam grandes programas a partir de componentes de software criados separadamente (AHMED; BAGHERZADEH, 2018). Após otimizar o mutex dentro de um componente de software, não é garantido ao programador que o desempenho do componente otimizado seja escalado após ele ser composto com outros componentes em um programa paralelo. Os mutexes também dificultam o fornecimento de garantias de segurança de exceções; um programador deve liberar cuidadosamente os mutexes corretos, na ordem correta (PANKRATIUS; ADL-TABATABAI, 2014).

2.4 Memória Transacional

A **Memória Transacional** (*Transactional Memory*, ou TM) é um mecanismo de sincronização alternativo aos mostrados na seção 2.1 que tem como base, para garantir sincronismo entre threads concorrentes, o conceito de transação. Uma transação consiste em uma sequência de instruções com atomicidade e isolamento, garantindo consistência no acesso a dados compartilhados. Durante sua execução, uma transação armazena localmente os acessos de leitura e escrita feitos aos dados compartilhados. Um conflito ocorre quando duas transações realizam operações escrita-escrita ou escrita-leitura no mesmo dado. 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; CENTODUCATTE; BALDASSIN, 2007).

A implementação do mecanismo de Memória Transacional pode ser realizada exclusivamente em software, conhecida como Memória Transacional em Software, ou

contar com suporte em hardware, conhecida como Memória Transacional em Hardware. Abordagens híbridas, que empregam ao mesmo tempo Memória Transacional em Software e Memória Transacional em Hardware, também existem.

A **Memória Transacional em Software** (*Software Transactional Memory*, ou STM) é um mecanismo de controle de concorrência em nível de software com o qual programadores podem particionar o código em transações e garantir que executem atomicamente e isoladamente em relação uma a outra. Uma transação é confirmada (realiza o *commit*) ou aborta, caso entre em conflito com outras transações, revogando quaisquer efeitos causados. A STM pode simplificar a programação concorrente, facilitando a proteção de dados, permitindo raciocínio sequencial entre transações e desabilitando parte dos bugs de concorrência. A Memória Transacional em Software oferece algumas vantagens sobre a Memória Transacional em Hardware (LARUS; RAJWAR, 2006): (i) O software é mais flexível, permitindo a implementação de uma grande variedade de algoritmos mais sofisticados; (ii) O software é mais fácil e barato de se modificar do que o hardware; (iii) Podem ser integradas mais facilmente com sistemas existentes e características de linguagens; (iv) Possuem menos limitações intrínsecas impostas por estruturas de hardware de tamanho fixo, como caches.

A **Memória Transacional em Hardware** (*Hardware Transactional Memory*, ou HTM), embora de emprego menos flexível que a memória transacional em software, pode oferecer vantagens chave (HARRIS; LARUS; RAJWAR, 2010): (i) possuem um overhead menor do que Memórias Transacionais em Software; (ii) podem ser menos invasivas nos sistemas atuais; por exemplo, Memórias Transacionais em Hardware que são implícitas podem garantir as propriedades de transação para bibliotecas de terceiros que não foram escritas para sistemas transacionais.

Memória Transacional em Software e Memória Transacional em Hardware devem executar da mesma maneira: devem identificar acessos à memória dentro de transações, gerenciar os conjuntos de leitura e escrita da transação, detectar e resolver conflitos, efetivar e abortar transações.

2.4.1 Transações

Uma transação consiste em uma sequência de instruções, incluindo leituras e escritas na memória, que executa completamente (realiza *commit*) ou não tem efeito (*abort* – é abortada). Quando uma transação realiza *commit*, todas as suas escritas são visíveis e os valores podem ser usados por outras transações. Quando uma transação é abortada, todas as suas escritas especulativas são descartadas. *Commit* ou *abort* são decididos com base na detecção de conflitos de memória entre transações paralelas. Elas são regidas por quatro propriedades básicas, conhecidas como propriedades ACID (HARRIS; LARUS; RAJWAR, 2010):

- **Atomicidade**: exige que todas as ações constituintes em uma transação sejam concluídas com sucesso ou que nenhuma dessas ações seja efetivamente executada. Não é aceitável que uma ação constituinte falhe e que a transação seja concluída com sucesso. Também não é aceitável que uma ação fracassada deixe provas de que executou.

- **Consistência**: visa garantir que as modificações geradas por uma transação não levem o sistema a um estado incorreto, ou seja, garante que o sistema sempre esteja em um estado válido.

- **Isolamento**: a execução da sequência de instruções de uma transação pode ser intercalada com a execução de instruções de outras transações, sem comprometer o resultado. Garante que as transações não irão interferir umas com as outras enquanto estão executando.

- **Durabilidade**: garante que os resultados gerados sejam permanentes e estejam disponíveis para as próximas transações.

A propriedade de durabilidade atenta ao fato de que as modificações de uma transação finalizada com sucesso devem ser permanentes (em um disco rígido, por exemplo). Esta qualidade – aplicável aos sistemas banco de dados, onde os dados são armazenados em memória persistente – não é aplicável às Memórias Transacionais pela própria natureza dinâmica dos dados na memória.

Os sistemas de Memória Transacional exigem mecanismos para gerenciar as tentativas de escrita que as transações concorrentes estão fazendo. É preciso que haja algum mecanismo para gerenciar essas tentativas, pois em um dado momento, existirão diversas versões dos dados: a versão antiga e a versão modificada por uma transação. Como várias transações podem acessar o mesmo dado, existem duas abordagens gerais para o **versionamento** (HARRIS; LARUS; RAJWAR, 2010), (LARUS; RAJWAR, 2006):

- **Versionamento Adiantado**: uma transação modifica o dado diretamente na memória e guarda o valor antigo em um *undo-log*; se ela abortar, o valor será escrito de volta. Assim, há necessidade de um controle de concorrência pessimista, pois o bloqueio de um local é necessário, já que a transação vai escrever nele diretamente;

- **Versionamento Preguiçoso**: as tentativas de escritas são armazenadas em um *redo-log* que é privado à transação e são atualizadas na memória apenas se e quando a transação efetiva. Se ela abortar, o *redo-log* é simplesmente descartado.

O versionamento adiantado é mais rápido no caso em que a transação realiza o *commit*, pois o *undo-log* pode ser descartado e nenhuma ação posterior deve ser tomada. Porém, se ocorre um *abort*, é necessário tempo adicional para desfazer as modificações. O versionamento preguiçoso, por sua vez, favorece o caso de ocorrência de *abort*, pois se uma transação for abortada, simplesmente descarta seu *redo-log*.

2.4.2 Controle de Concorrência

Sobre controle de concorrência, de forma geral, existem dois mecanismos principais que uma implementação de memória transacional precisa prover (HARRIS; LARUS; RAJWAR, 2010): (i) garantia de isolamento entre transações, detectando e resolvendo possíveis conflitos para que elas pareçam estar executando sequencialmente (serialização); e (ii) gerenciamento do trabalho que uma transação faz enquanto ela executa.

Acessos a dados podem causar conflitos. Existem três eventos relacionados a esses conflitos, que podem ocorrer em tempos diferentes, mas não em ordens diferentes (HARRIS; LARUS; RAJWAR, 2010): **Ocorrência**: um conflito ocorre quando duas transações realizam operações escrita-escrita ou escrita-leitura no mesmo dado; **Detecção**: a detecção ocorre quando o sistema de Memória Transacional determinar que o conflito ocorreu; e **Resolução**: o conflito é resolvido quando o sistema de Memória Transacional realizar alguma ação para evitá-lo.

Como o versionamento, a detecção de conflitos tem duas abordagens: adiantada (pessimista) e ou preguiçosa (otimista) (GUERRAOUI; KAPALKA, 2010), (LARUS; RAJWAR, 2006):

- **Detecção Adiantada**: ocorre no momento em que uma transação realiza acesso a memória (leitura ou escrita).
- **Detecção Preguiçosa**: conflitos são detectados somente quando a transação tenta efetivar (commit), ou seja, no final da execução da transação.

O versionamento de dados adiantado necessita que a detecção de conflitos também seja adiantada, a fim de evitar inconsistência nos dados computados pelas transações. Porém, o versionamento de dados preguiçoso não obedece a esta regra, podendo usufruir de ambos os tipos de detecção de conflitos. Para resolver um conflito, as implementações de Memórias Transacionais utilizam um gerenciador de contenção (*Contention Management*) (GUERRAOUI; HERLIHY; POCHON, 2006), que possui a finalidade de decidir qual transação deve ser abortada ou reiniciada em função de regras específicas onde, dependendo do tipo de regra adotada pelo gerenciador, pode-se conseguir melhor desempenho por diminuir a ocorrência de *aborts* desnecessários.

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; KAPAŁKA, 2010). Em (HARRIS; LARUS; RAJWAR, 2010) é apresentada uma interface de operações transacionais que podem ser suportadas tanto por Memória Transacional em Hardware quanto por Memória Transacional em Software:

- `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.

Pode-se apresentar o uso de Memória Transacional através de um exemplo com operações em agências bancárias. Supondo que exista um array denominado `conta`, no qual cada entrada `conta[i]` é o valor armazenado na conta de número i, com um procedimento simples para o depósito de um valor em uma determinada conta `deposita_conta (i, valor)`: soma a quantia valor à `conta[i]`;

Um mesmo número de conta pode ser acessado por mais de um cliente do banco ao mesmo tempo; portanto, um mecanismo de sincronização necessita estar presente. Uma possível implementação para o procedimento, utilizando *locks* é mostrado na Figura 1.

O mesmo código, utilizando Memória Transacional para o procedimento é mostrado na Figura 2, onde `StartTx()` e `CommitTx()` delimitam o escopo da transação.

O código mostrado na Figura 3 exemplifica o mesmo trecho de código, agora delimitado por um bloco atômico, responsável por identificar o início e o fim de uma

```
1                       deposita_conta (i, valor) {
2                           lock (conta[i]);
3                           conta[i] = conta[i] + valor;
4                           unlock (conta[i]);
5                       }
```

Figura 1 – Exemplos diretiva `critical` - OpenMP

```
1                       deposita_conta (i, valor) {
2                           do {
3                               StartTx();
4                               valor_atual = ReadTx(conta[i]);
5                               WriteTx(conta[i], valor_atual + valor);
6                           } while (!CommitTx());
7                       }
```

Figura 2 – Exemplos diretiva `critical` - OpenMP

transação. o compilador se encarrega de substituir `atomic{}` pelas operações delimitadoras e introduzir as operações `ReadTx` e `WriteTx` onde for necessário.

```
1                       deposita_conta (i, valor) {
2                           atomic {
3                                   conta[i] = conta[i] + valor;
4                                  }
5                       }
```

Figura 3 – Exemplos diretiva `critical` - OpenMP

O código fonte com a utilização de Memórias Transacionais torna-se extremamente simples. Todo o controle de concorrência é realizado pelo sistema de memória transacional. A construção `atomic` representa o bloco atômico, responsável por delimitar o escopo que deve ser executado por uma transação. O sistema transacional é responsável por garantir a sincronização do bloco. O programador apenas especifica quais serão os blocos atômicos e não em como sincronizá-los.

2.4.3 Vantagens do uso de Memória Transacional

Mutexes, como exposto anteriormente, são complexos de usar e propensos a erros, especialmente quando um programador está em busca de uma melhor escalabilidade de seu programa em hardware altamente paralelo usando bloqueio de granularidade fina. Uma das razões é que a funcionalidade de sincronização provida pelos mutexes não está associada ao dado compartilhado em si, mas sim associada ao controle do avanço das execuções das instruções de cada fluxo envolvido.

A Memória Transacional pode ser utilizada para este problema, abstraindo as complexidades associadas ao acesso simultâneo aos dados compartilhados. Sua

principal característica é vincular a sincronização ao(s) dado(s) manipulado(s), realizando a validação da comunicação, representada pela escrita ou leitura de dados compartilhados, quando a operação se der sem conflitos. O modelo de sincronização oferecido por Memória Transacional se apresenta como alternativa aos métodos baseados em mutexes, oferecendo maiores nível de abstração na programação no uso de operações atômicas. Também, devido às características de atomicidade e isolamento, sistemas transacionais podem explorar mais paralelismo, aumentando sua escalabilidade e seu desempenho. A Memória Transacional oferece as seguintes vantagens (RIGO; CENTODUCATTE; BALDASSIN, 2007):

- **Escalabilidade**: transações que acessem um mesmo dado para leitura podem ser executadas concorrentemente. Essa característica dinâmica permite que mais desempenho seja obtido com o aumento do número de processadores (ou threads) porque o nível de paralelismo exposto é maior.

- **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. O conceito de robustez apresentado na proposta tem origem a partir desta vantagem.

- **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). Para isso, estratégias de aninhamento devem ser implementadas. Novamente, o sistema de execução garante que as operações sejam executadas de forma atômica.

Alguns dos principais benefícios da Memória Transacional, em comparação com os mutexes são descritos em (BOEHM et al., 2012). Os resultados obtidos neste trabalho foram coletados considerando estudos realizados na linguagem C++. Em linhas gerais, os principais benefícios levantados são apresentados na sequência.

- Em estudos de caso, a Memória Transacional demonstrou reduzir o desafio da programação paralela em comparação com a exclusão mútua (PANKRATIUS; ADL-TABATABAI, 2011), (ROSSBACH; HOFMANN; WITCHEL, 2010a). Embora o conceito de exclusão mútua seja simples, os programas que usam principalmente mutexes podem rapidamente tornar-se complexos, mesmo para programadores paralelos experientes. Os estudos mencionados mostraram que o uso de transações em vez de mutexes resultou em programas mais simples, resultando em significativa redução no número de erros de programação.

- A Memória Transacional evita a ocorrência de situações de postergação indefinida, já que funções que usam apenas sincronização transacional podem ser compostas sem risco de interblocagem.

- Memória transacional suporta um modelo de programação modular; isto é, um modelo de programação onde o software é escrito compondo módulos separados e intercambiáveis. A programação modular é fundamental para o C++. Todas as bibliotecas padrão do C++ e seu design de linguagem de programação genérica suportam um modelo de programação modular. Em contraste com os problemas com mutexes, descritos na Seção 2.3, a Memória Transacional é composável.

- A Memória Transacional aumenta o nível de abstração em comparação à exclusão mútua porque as transações suportam uma gama mais ampla de abordagens de implementação, incluindo a execução especulativa. Quando os programadores utilizam transações para controlar o acesso a dados compartilhados, eles especificam o que é sincronizado, não como é sincronizado. Isso permite um maior grau de liberdade de implementação para produzir soluções mais eficientes.

- A Memória Transacional pode ser projetada de forma que possa ser usada em conjunto com a maioria dos mecanismos de concorrência de C++ existentes e, nos casos em que não puder, pode ser projetada de tal forma que incompatibilidades de concorrência sejam identificadas como erros em tempo de compilação.

Diferente do mecanismo baseado em bloqueios imposto pelo uso de mutex, o modelo de Memória Transacional explora o conceito de transações atômicas. Um programador define uma transação colocando um conjunto de instruções da linguagem de programação dentro de um bloco atômico. Esse bloco representa uma seção crítica e deve conter apenas declarações com efeitos reversíveis. Um *runtime system* permite que os threads executem blocos atômicos concorrentemente, fazendo parecer que apenas um thread por vez é executado dentro de um bloco atômico. Se uma transação executando concorrentemente entra em conflito com outra transação, o *runtime* aborta (ou seja, desfaz seus efeitos) e tenta novamente mais tarde; caso contrário, ele o confirma (realiza o *commit*) e torna seus efeitos visíveis para todos os outros threads. O *runtime* basicamente impõe a atomicidade, consistência e propriedades de isolamento conhecidas de transações de banco de dados que se aplicam a instruções de linguagem de programação (PANKRATIUS; ADL-TABATABAI, 2014).

2.4.4 Desafios da Utilização da Memória Transacional

Apesar dos recentes avanços na pesquisa da Memória Transacional, há pouca experiência no uso de TM para desenvolver programas paralelos mais realistas do zero. Discussões como (PANKRATIUS; ADL-TABATABAI, 2011), (ROSSBACH; HOFMANN; WITCHEL, 2010b) e (PANKRATIUS; ADL-TABATABAI, 2014) sobre Memória Transacional versus mutexes, focam em pequenos programas ou microbenchmarks para avaliar o caso de pior desempenho, mas nenhum deles leva em conta aplicações mais complexas para avaliar aspectos de engenharia de software por um longo período de tempo. Outros trabalhos estudando a conversão de programas baseados em mutexes para TM não lançaram luz sobre os problemas encontrados ao paralelizar com MT a partir do zero (PANKRATIUS; ADL-TABATABAI, 2011).

Embora Memória Transacional de Software (STM) seja uma técnica promissora para alcançar concorrência mais fácil/segura e evitar bugs de concorrência, apresenta desvantagens que dificultam sua adoção na prática (YU; ZUO; XIONG, 2019). Muitos dos fatores limitantes são reflexos do chamamo código legado, ou seja, do código já existente, desenvolvido sem pensar nos seus custos futuros de manutenção.

A primeira limitação pode ser atribuída ao alto custo humano para portar programas já desenvolvidos com funcionalidade de transação. A maioria dos sistemas de Memória Transacional de Software são implementados como bibliotecas de programação com APIs para uso. Para mover o código baseado em mutexes para baseados em transações, os programadores precisam verificar o código com cuidado e inserir chamadas de API destas bibliotecas nos pontos adequados. A dificuldade em utilizar uma biblioteca de serviços está associada ao fato de que não existe uma compatibilidade direta de tipos manipulados no programa e suportados pelas bibliotecas (BOEHM, 2005), retirando, parcialmente, a responsabilidade do compilador pela verificação do código e, em consequência, reduzindo sua confiabilidade. As chamadas para APIs da Memória Transacional de Software são utilizadas para demarcar transações e identificar possíveis acessos compartilhados nelas. Além disso, o emprego de transações também requer alterações nas estruturas de dados/controle. Embora exista métodos propostos (FENG; GUPTA; NEAMTIU, 2013) para aliviar esse ônus, eles não removem tudo.

A segunda limitação está associada à baixa compatibilidade com chamadas de E/S. Enquanto o espaço de memória do usuário dentro de uma transação está sob o controle do sistema da Memória Transacional de Software, a memória no kernel pode não estar. Como resultado, as propriedades de atomicidade e isolamento não são aplicadas automaticamente para alterações de estruturas de dados no kernel. Além disso, algumas operações de E/S, como imprimir uma mensagem na tela, podem não serem revertidas ao abortar. Portanto, a maioria das Memórias Transaci-

onais de Software, como Grace (BERGER et al., 2009) e Haskell STM (HARRIS et al., 2005b), simplesmente proíbe o uso de chamadas de E/S nas transações. Análises de programas multithread escritos com mutexes mostram que as chamadas de E/S são uma ocorrência regular em seções críticas (VOLOS et al., 2009), (Baugh; Zilles, 2008). Além disso, um estudo de bugs de concorrência mostra que cerca de 15% da concorrência a recuperação de bugs envolve a revogação de efeitos de chamadas de E/S (VOLOS et al., 2012b). Portanto, proibir chamadas de E/S em transações reduz a usabilidade da Memória Transacional de Software e ameaça sua validade como uma solução para bugs de concorrência.

2.5 OpenMP

Esta seção apresenta OpenMP, explorada, neste trabalho como base de extensão para implementação da interface proposta.

2.5.1 Contextualização da Ferramenta

A especificação de OpenMP (*Open Multi-Processing*) (DAGUM; MENON, 1998), (OpenMP Architecture Review Board, 2011) define uma API, para explorar a concorrência em programas C e Fortran, tendo como objetivo definir uma ferramenta de programação multithreaded com alto grau de portabilidade. OpenMP é voltada originalmente para ambientes de memória compartilhada e sua API inclui a especificação de variáveis de ambiente, de uma biblioteca de serviços e de um conjunto de diretivas de compilação. Estas diretivas de compilação permitem descrever a concorrência de um programa e definir a sincronização entre atividades concorrentes. Da forma como foi concebida, a indicação de trechos concorrentes no código é realizada de forma explicita, anotando um código original com as diretivas especificadas. A exploração do paralelismo, no entanto, se dá de forma implícita, implementando um mecanismo de escalonamento em nível aplicativo sobre um modelo de execução $n \times m$.

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.

No modelo de programação oferecido pelo OpenMP, os threads podem possuir uma visão temporária (de parte) da memória de forma a evitar acessos sistemáticos à memória compartilhada e também uma porção de memória privada que contém

cópia de uma parte, explicitada no programa, da memória compartilhada. Não é previsto um ordenamento global dos acessos a nenhum dos dois níveis de memória. Assim, cabe ao programador introduzir, explicitamente, mecanismos de controle de acesso tanto dos threads à memória compartilhada do processo, quanto das tarefas à memória privada do thread sobre o qual está executando. O que é garantido é que, embora sem conhecimento do ordenamento de ações, várias diretivas de OpenMP, quando executadas, promovem de maneira implícita, a atualização dos diferentes níveis de memória com os valores contidos nas instâncias dos dados que estão sendo manipulados pelas tarefas e/ou threads naquele momento.

Desde sua primeira versão, datada de 1998, OpenMP manteve-se atualizado, 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. A partir da versão 3.0 (CHAPMAN; JOST; PAS, 2007), foi incorporado um meio para expressão explícita de tarefas, aumentando seu poder de expressividade. Regiões paralelas, na nomenclatura OpenMP, definem seções do código em que a concorrência é apresentada de forma explícita. Os termos tarefa e thread são utilizados para representar, respectivamente, thread usuário e thread sistema, n e m, respectivamente, no modelo $n{\times}m$. A concorrência descrita nestas regiões paralelas permite a criação de tarefas, que são manipuladas pelo núcleo de escalonamento OpenMP e escalonadas sobre os threads disponíveis. O conjunto destes threads representa um time de execução. Por padrão, o número de threads desse time corresponde ao número de *cores* da máquina onde ocorre a execução, podendo ser alterado dinamicamente. Os threads representam *processadores virtuais* oferecidos pelo núcleo de execução de OpenMP, consumindo, ou seja, executando, tarefas instanciadas na região paralela. O modelo básico de execução é caracterizado como *fork/join*. Uma tarefa, em execução, pode instanciar novas tarefas (*fork*), sendo suspensa até que as tarefas criadas concluam (*join*). Podendo, este comportamento ser aninhado (*nested fork/join*) (CHANDRA et al., 2001).

Um time de threads é criado para executar o código em uma região paralela de um programa OpenMP. Para isso, o programador especifica a região paralela inserindo uma diretiva `parallel` antes do código que deve ser executado em paralelo para marcar o seu início. Informações adicionais podem ser fornecidas junto com a diretiva `parallel`. Isso é usado principalmente para permitir que os threads tenham cópias particulares de alguns dados para a duração da região paralela, para inicializar esses dados e para copiar o último valor também. No final de uma região paralela há uma sincronização implícita de barreira: isso significa que nenhum thread pode progredir até que todos os outros threads do time tenham atingido esse ponto no programa. Após isso, a execução do programa continua com o thread ou

threads que existiam anteriormente. Se um time de threads executando uma região paralela encontrar outra diretiva `parallel`, cada thread no time atual cria um novo time de threads e se torna seu mestre. O aninhamento permite a realização programas paralelos de vários níveis (CHAPMAN; JOST; PAS, 2007).

2.5.2 Descrição da Concorrência

A interface de programação de OpenMP tem nas suas diretivas o elemento básico para descrição da concorrência de um programa e introdução de pontos de sincronização. Diretivas devem ser utilizadas em uma região paralela delimitada por `parallel` e possuem a forma[1]: `#pragma omp <diretiva> [cláusulas]`. Nesta sintaxe, `#pragma omp` instrui 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. As cláusulas para as diretivas são opcionais e permitem definir o acesso de cada tarefa aos identificadores que se encontram no escopo das tarefas geradas.

Duas das principais diretivas de paralelização de OpenMP são `for` e `task`. No padrão OpenMP, a construção de laço (*loop*) `for` faz com que as iterações do laço imediatamente após ele sejam executadas em paralelo. No tempo de execução, as iterações do laço são distribuídas entre os threads. Em outras palavras, a diretiva `for` divide o trabalho do laço `for` entre os threads do time atual. A diretiva `#pragma omp parallel for` é um atalho para se utilizar ambos – `parallel` e `for` – em uma só diretiva: enquanto `parallel` cria um novo time de threads, `for` divide o time para manipular diferentes porções do laço. A diretiva `task` pode ser usada para definir explicitamente uma tarefa. É utilizada quando se deseja identificar um bloco de código a ser executado em paralelo com o código fora da região de `task` (CHAPMAN; JOST; PAS, 2007).

2.5.3 Modelo de Memória

No modelo de memória do OpenMP, uma variável pode ser do tipo compartilhada (`shared`) ou do tipo privada (`private`). Variáveis privadas são de uso específico de cada thread. 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. Portanto, **diretivas de sincronização** garantem que o acesso ou atualização de uma determinada variável compartilhada aconteça no momento certo. Essas diretivas são listadas a seguir (CHAPMAN; JOST; PAS, 2007) (CHANDRA et al., 2001):

- **`critical`**: fornece um meio para garantir que vários threads não tentem atu-

[1] Neste trabalho, limitamos a discussão da sintaxe ao padrão OpenMP para a linguagem C/C++.

alizar os mesmos dados compartilhados simultaneamente. É utilizado para evitar que uma mesma variável compartilhada seja atualizada por mais de um thread ao mesmo tempo. Quando um thread encontra uma diretiva `critical`, ele aguarda até que nenhum outro thread esteja executando as instruções da seção crítica de mesmo nome. Quando nenhum thread estiver executando essa seção crítica, o thread que estava esperando entra na região e executa as instruções.

- **`atomic`**: permite a atualização eficiente de variáveis compartilhadas por múltiplos threads em plataformas de hardware que suportam operações atômicas. Impede que vários threads acessem uma região da memória, permitindo que essa região seja atualizada atomicamente. Habilita múltiplos threads a atualizarem dados compartilhados sem interferência. Diretiva utilizada para proteger uma atualização única para uma variável compartilhada, sendo usada para uma única instrução. Essa restrição é aplicada a todos os threads que executam o programa, não somente os threads pertencentes a um mesmo time.

- **`barrier`**: sincroniza todas os threads em um determinado ponto do código. Permite ao programador criar uma barreira de sincronização explícita, a qual os threads só podem seguir seus fluxos de execução após todos do mesmo grupo passarem por ela, a partir daí, continuam a execução do código ao mesmo tempo.

- **`ordered`**: permite executar um bloco estruturado dentro de um loop paralelo em ordem sequencial. Isso às vezes é usado, por exemplo, para impor uma ordem na impressão de dados calculados por diferentes threads.

- **`master`**: define um bloco de código que será executado apenas pelo thread mestre. Quando se utiliza esta diretiva, o trecho compreendido só pode ser executado pelo mestre do grupo, ou seja, o thread que recebe o identificador zero.

- **`flush`**: garante que todos os threads tenham acesso ao valor correto de uma variável compartilhada que foi recentemente atualizada. O padrão OpenMP especifica que todas as modificações em variáveis compartilhadas sejam escritas na memória principal em pontos de sincronização específicos, dessa forma, tornando-se acessíveis a todos os threads. Porém, nas regiões localizadas entre os pontos de sincronização, não há garantia de que as variáveis sejam atualizadas instantaneamente na memória principal. Para garantir essa atualização nos trechos entre os pontos de sincronização, utiliza-se a diretiva `flush`. Quando uma variável compartilhada é atualizada por um thread, o novo valor dessa variável fica inicialmente armazenado somente na memória cache do

processador, pois o acesso a essa memória é bem mais rápido que o acesso à memória principal.

As diretivas **atomic** e **critical** podem ser vistas como implementações de mecanismos de exclusão mútua em OpenMP, pois ambas diretivas fornecem meios para garantir que vários threads não atualizem simultaneamente os mesmos dados compartilhados em uma região de memória, permitindo que essa região seja atualizada atomicamente.

A diretiva **atomic** é aplicada a uma única instrução de atribuição que atualiza uma variável escalar, a qual segue a declaração desta diretiva. O acesso atômico é garantido em uma única instrução de baixo nível, não em toda a instrução de alto nível apresentada. Observando os dois exemplos seguintes do uso da diretiva `atomic`, na Figura 4 pode-se reconhecer a semântica do acesso atômico realizado. No primeiro exemplo, a semântica é de um acesso em incremento atômico na variável identificada por `x`, sendo a operação atômica realizada sobre a operação de baixo nível `inc`. No segundo, a operação realizada de forma atômica é o `mov` do valor resultante da expressão `a + b` para a variável identificada por `x`. Ou seja, os acessos às variáveis `a` e `b` não são realizados de forma atômica, nem a operação de soma.

```
1  #pragma omp atomic
2      x++;
3  #pragma omp atomic
4      x = a + b;
```

Figura 4 – Exemplos diretiva `atomic` – OpenMP

A diretiva **critical** possui uma granularidade maior, sendo aplicada a uma instrução, que pode ser também uma instrução composta (bloco de comandos). Dois exemplos do uso deste comando são mostrados na Figura 5:

```
1  #pragma omp crittical
2     x++;
3  #pragma omp critical SOMA
4     x = a + b;
```

Figura 5 – Exemplos diretiva `critical` - OpenMP

No primeiro exemplo, a semântica obtida é idêntica àquela obtida no primeiro exemplo do uso do `atomic`. A diretiva usada desta forma considera o conjunto de instruções que acompanha uma seção crítica globalmente sincronizada. No segundo exemplo, a instrução que acompanha a diretiva é realizada em regime de exclusão mútua com toda e qualquer outra seção crítica acompanhada do mesmo rótulo `SOMA` informado.

As diretivas do OpenMP suportam várias **cláusulas**, adições opcionais que fornecem uma maneira simples e poderosa para controlar o comportamento do construtor o qual elas se aplicam. De fato, algumas dessas cláusulas são quase indispensáveis na prática. Elas incluem a sintaxe necessária para especificar quais variáveis são compartilhadas e quais são privadas no código em execução de cada tarefa. Diferentes cláusulas estão disponíveis, oferecendo semânticas que atendem a diferentes modos de acesso. Sabendo que um identificador em um programa refere-se a um endereço de memória em que está armazenada a variável compartilhada, as cláusulas definem como será feito o acesso às variáveis, via identificadores, nas unidades de código concorrente. As seguintes cláusulas oferecem semânticas de acesso a dados, sendo que elas recebem uma lista de identificadores, cujo escopo é válido, para variáveis: (CHAPMAN; JOST; PAS, 2007).

- **`shared`** (CRCW – *Concurrent Read, Concurrent Write*): Sem garantias de sincronização ao acesso implícito, todas tarefas veem a modificação que todos fazem e sobre quais dados, sem uso de mecanismos de sincronização explícitos, como o `critical` ou o `atomic`, podem incorrer condições de corrida. As variáveis `shared` encontram-se armazenadas no espaço de endereçamento compartilhado pela tarefa master.

- **`private`** (EREW – *Exclusive Read, Exclusive Write*): Oferece uma semântica de acesso sequencial, não concorrente. Cada tarefa possui uma instância própria da variável indicada e realiza sobre ela seu trabalho.

- **`firstprivate`** (EREW – *Exclusive Read, Exclusive Write*): Idem a anterior, mas permite que a instância privada a cada tarefa seja inicializada com o valor anotado na tarefa master. Define uma lista de variáveis com o atributo `private`, mas sendo inicializadas automaticamente, de acordo com o valor que possuíam no programa (thread mestre) antes de uma região paralela.

- **`lastprivate`** (ERCW – *Exclusive Read, Concurrent Write*): Também semelhante a cláusula `private`, cada tarefa possui uma instância própria da variável e sobre ela trabalha. A semântica de escritas concorrentes oferecida na cláusula `lastprivate` garante que a variável original na tarefa master será sobre-escrita com o valor anotado na instância local à tarefa que for executada por último. Um mesmo identificador pode ser passado como `first` e `lastprivate` simultaneamente às tarefas criadas.

- **`reduction`** (CRCW – *Concurrent Read, Concurrent Write*): A semântica desta cláusula permite que resultados de cálculos parciais, realizados de forma independente pelas tarefas, seja *reduzido* em uma informação única no término de

todas as tarefas associadas. Cada tarefa possui uma instância local da variável indicada, inicializada com o valor identidade da operação de redução aplicada. A semântica de escrita concorrentes diz que a variável original na tarefa master será o resultado da redução aplicada sobre os dados produzidos nas instancias locais a cada tarefa filha da tarefa master.

Ainda, neste contexto, deve ser citada a diretiva **`threadprivate`**. Esta diretiva recebe como parâmetro uma lista de identificadores, os quais passam a representar variáveis compartilhadas entre os threads do time de execução. A semântica de acesso à estas variáveis é idêntica ao uso da cláusula `shared`. No entanto há duas ressalvas: (*i*) as variáveis neste caso são inicializadas com valor indeterminado, e (*ii*) estas variáveis não são compartilhadas entre threads de diferentes times de execução.

As cláusulas com semânticas mais simples são `shared` e `private`. Ao utilizar a cláusula `shared` para um determinado identificador, é informado que, em todas as tarefas criadas, o identificador deve continuar se referenciando à variável original. Todo acesso, portanto, carece do uso de mecanismos extras para controle de condições de corrida, como o uso de `critical`, `atomic` ou `ordered`. A programação, portanto, torna-se mais difícil, pois o programador precisa se preocupar em como garantir a sincronização. Desse modo, a responsabilidade pela sincronização é toda do programador.

Já ao utilizar a cláusula `private` é informado que cada tarefa deve ter uma instância própria da variável indicada pelo identificador passado como parâmetro e que, nas tarefas, este identificador passa a se referenciar a instância da variável local. As instâncias locais tem valor inicial indeterminado e sua manipulação uma tarefa não é refletida nas instâncias locais às outras tarefas. Não é necessário, portanto, nenhum outro mecanismo de controle de acesso.

As cláusulas `firstprivate` e `lastprivate` são bastante similares à `private`, sendo que `firstprivate` difere por inicializar as instâncias locais às tarefas com o último valor anotado no dado original e que `lastprivate` permite que a variável original seja atualizada, ao final da execução de todas as tarefas associadas a essa cláusula, com o valor anotado na instância local à tarefa que tenha terminado sua execução por último. Nada impede que um determinado identificador esteja listado como `firstprivate` e `lastprivate` em uma mesma diretiva.

A cláusula `reduction` oferece uma semântica de leituras e escritas concorrentes baseada em uma operação de redução que possui a propriedade de ser associativa e comutativa. Em OpenMP existe uma lista de operações de redução disponíveis, sendo que a operação desejada sobre uma determinada variável deve ser informada junto com seu identificador. Esta cláusula faz com que exista uma instância local a cada tarefa da variável original, sendo esta inicializada com o valor identidade

da operação de redução. Cada tarefa manipula sua instância local sem que isso afete a computação das demais tarefas. Ao final da execução de cada tarefa, o dado contido na instância local é reduzido, pela operação indicada, no dado original. Ao utilizar a cláusula `reduction`, nenhum outro mecanismo de controle de acesso se faz necessário, embora o programador se restrinja a utilizar as operações de redução disponíveis.

O OpenMP possui ainda uma biblioteca com rotinas de interface de programação que possibilitam uma grande variedade de funções. No que diz respeito à sincronização, as seguintes primitivas permitem utilizar um mecanismo baseado em mutex, identificado como *lock* na nomenclatura OpenMP:

- **`omp_init_lock`**: inicia as variáveis de bloqueio, indicando para os threads as variáveis de bloqueio. A variável tem que ser definida com tipo *lock*.
- **`omp_destroy_lock`**: finaliza qualquer associação de bloqueio de uma determinada variável.
- **`omp_set_lock`**: solicita o bloqueio de uma variável, ou aguarda que uma variável bloqueada esteja disponível.
- **`omp_unset_lock`**: libera o bloqueio de uma determinada variável.
- **`omp_test_lock`**: testa e bloqueia, se uma variável de bloqueio, está disponível para ser bloqueada, mas sem parar a execução do thread.

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, é necessário coordenar o acesso a variáveis compartilhadas por múltiplos threads, a fim de garantir a execução correta. Isso, em um programa OpenMP, é feito tanto pelo uso dos mecanismos de sincronização disponíveis, como, e sobretudo, utilizando as cláusulas de parametrização das tarefas.

2.5.4 Mapa de Memória

A Figura 6 ilustra um código OpenMP (a) e o mapa de memória (b) obtido durante sua execução. Neste programa exemplo, existem quatro threads no time de execução e, por consequência, a diretiva `task` instancia quatro tarefas para executar o código de seu bloco. Considere que a imagem do mapa de memória representa um momento em que todas as quatro tarefas foram instanciadas e estão em execução. Na imagem, há destaque para quatro estratos de memória. No alto é apresentado o escopo dos identificadores globais ao módulo. Os estratos seguintes representam, na ordem, o escopo de identificadores compartilhados entre os threads do time de

execução e o escopo definido no contexto da função `main`. Por fim, o último estrato identifica o escopo local a cada tarefa.

Na ilustração apresentada, os identificadores `g` e `h` são globais ao módulo, com variáveis instanciadas na área de memória estática do processo. As variáveis identificas por `a` e `b` são instanciadas na pilha da função main. A diretiva `threadprivate` e as cláusulas `private` e `shared` determinam como estas variáveis serão observadas pelos threads e tarefas instanciadas. Em um primeiro momento, aponta-se que ao identificador `g` não foi associada nenhuma semântica de acesso por OpenMP. Estando este identificador declarado global ao módulo, e a variável correspondente alocada na área de memória estática, as tarefas podem acessá-lo diretamente, observando, para evitar condições de corrida, mecanismos de sincronização adequados.

O identificador `h` também foi declarado global ao módulo. No entanto, o uso da diretiva `threadprivate` instancia uma nova variável cujo acesso é compartilhado entre os threads do time de execução. Portanto, havendo outro time de execução ativo, não sendo ilustrada esta situação no exemplo, não haveria compartilhamento de dados. Por outro lado, as tarefas compartilham acesso à essa nova instância, devendo também ser utilizados mecanismos de sincronização para garantir correção na execução.

Os identificadores `a` e `b` são passados por parâmetro às tarefas criadas. Enquanto a variável identificada por `b` é compartilhada entre todas as tarefas, devendo, portanto, seu uso ser controlado por mecanismos de sincronização, cada tarefa possui uma instância própria da variável identificada por `a`. Quando acessado o identificador `a`, a variável manipulada é aquela privada à tarefa. Quando os identificadores `h` e `b` são acessados, as variáveis manipuladas são, respectivamente, a que está na área de memória compartilhada entre os threads e na área de memória da pilha da função `main`.

2.5.5 Questões de Sincronização

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, `private(a)` e `shared(b)` correspondem a anotação de parâmetros para as tarefas, sendo `a`, por ser `private`, de entrada e `b`, por ser `shared`, de entrada e saída. Enquanto a semântica associada à manipulação `a` garante consistência no acesso ao dado em uma política de leituras e escritas concorrentes, de fato, havendo uma variável em memória diferente para

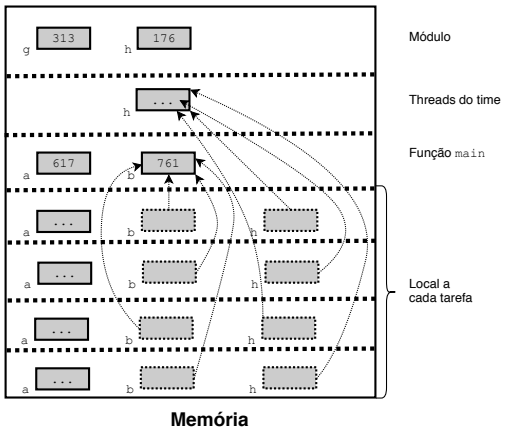
```
#include <omp.h>

int g = 313, h = 176;

int main( int argc, char **argv ) {
 int a = 617, b = 761;

 #pragma omp threadprivate(h)
 #pragma omp parallel num_threads(4)
 {
  #pragma omp task private(a), shared(b)
   {
    ...
    ...
    ...
   }
  #pragma omp taskwait
 }
 return 0;
}
```

(a)



(b)

Figura 6 – Exemplo de um programa OpenMP (a) e o mapa de memória obtido em tempo de execução.

o mesmo identificador, o mesmo não é garantido na manipulação de `b`, pois, neste caso, a variável acessada por todas tarefas é a mesma.

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 impasse, 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 impasse. 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 impasse e, em função de sua menor prioridade de execução, podem ocasionar contenção do paralelismo (SÜSS; 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.

2.5.6 Recursividade em OpenMP

Em OpenMP, algoritmos recursivos podem ser escritos utilizando a diretiva `task`. À esta diretiva é seguido um bloco de comandos, ou mesmo um comando simples, que é instanciado na forma de uma nova tarefa. Diferente do que ocorre por padrão com as demais diretivas, a nova tarefa passa a ser executada de forma concorrente com a tarefa criadora. A sincronização entre tarefas é explícita, com invocação à diretiva `taskwait`. Ao invocar uma sincronização, a tarefa suspende sua execução aguardando que todas as tarefas que ela própria tenha criado tenham terminado. Como uma tarefa pode criar, ela própria, novas tarefas, a concorrência obtida possui uma estrutura recursiva, com largura dependente do número de tarefas criadas por nível e profundidade dependendo de parâmetros próprios ao algoritmo. Como alternativa, a diretiva `taskgroup`, aplicada a um bloco de comandos, permite agrupar neste bloco a criação de tarefas, garantindo que a saída do bloco somente poderá ser realizada quando todas as tarefas criadas, incluindo as descendentes, tenham terminado.

A cláusula `if`, que é acompanhada de uma expressão aritmética, permite que o programador indique, dinamicamente, a granularidade desejada em seu programa. A expressão apresentada nesta cláusula permite determinar um limite a partir do qual novas tarefas não devem ser instanciadas, mas sua execução serializada na própria invocação. É possível assim otimizar a execução do código: caso o programador possa especificar uma expressão em que avalie se o custo de criação de uma tarefa compensa o seu custo computacional, Caso a avaliação da expressão retorne um valor falso, ou seja o valor inteiro zero (0) em C, a tarefa criadora é suspensa para que a nova tarefa possa executar imediatamente. A tarefa suspensa é retomada quando do término da nova tarefa.

As cláusulas de limitação de escopo permitidas com a diretiva `task` são: `shared`, `private` e `firstprivate`. Não estando, portanto disponíveis, as cláusulas `reduction` e `lastprivate`, não há nenhuma alternativa para o programador de controle de acesso implícito à memória. A manipulação de variáveis compartilhadas entre tarefas deve, portanto, ser realizadas em seções críticas de código, garantidas com uso de mecanismos de sincronização explícitos, seja pelo uso da diretiva `critical`, seja pelo uso da abstração para mutex provida pela biblioteca de serviços. Neste caso, considerando a natureza recursiva do algoritmo e sua execução concorrente, é possível que, por uma falha de concepção do programa, ocorram situações de impasse, devido a dependências cruzadas na obtenção do recurso de sincronização. Esta possibilidade deve ser evitada pelo próprio programador, no entanto, o próprio OpenMP oferece a diretiva `taskyield` para forçar a ativação do escalonamento, evitando a situação de impasse indesejada. A diretiva `taskgroup`, por sua vez, permite apenas o uso da cláusula `task_reduction`, oferecendo uma semântica de redução.

Também é possível, com a cláusula `depend( tipo : lista )` estabelecer uma relação de dependência de entrada e saída entre tarefas na forma de um DAG[2]. Nesta cláusula, `tipo` pode ser definido como `in`, `out` ou `inout` e `lista` a lista de identificadores sujeitos ao respectivo tipo de dependência. Neste caso, não é especificado um modo de acesso às variáveis pelas tarefas, mas sim especificada uma ordem em que os acessos devem ser realizados, observando-se uma relação de produção (`out`) e consumo (`in`) de dados entre elas.

As tarefas, ao serem criadas, possuem um atributo de amarração ao thread sobre o qual iniciaram sua execução. Por padrão, tarefas são amarradas (*tied*) ao thread sobre o qual são lançadas. Isso significa que, ocorrendo uma operação de escalonamento e elas sendo, ou estando, preemptadas, não poderão ser retomadas em outro thread que não aquele onde sua execução foi iniciada. Este padrão pode ser alterado pela cláusula `untied`, permitindo que a execução de uma tarefa seja retomada em qualquer thread, por decisão do escalonamento. Este atributo é determinante nas tomadas de decisão por parte do escalonador. O algoritmo de escalonamento, em linhas gerais, possui o seguinte comportamento (CAVALHEIRO; DU BOIS, 2014):

- As decisões de escalonamento são tomadas quando habilitados por operações próprias ao OpenMP, como na criação de tarefas, na execução de um `taskwait`, em uma barreira (implícita ou explícita), ou na invocação explícita do escalonamento pela diretiva `taskyield`.

- As ações de escalonamento conduzidas no núcleo de execução de OpenMP, que são aplicadas a todos os threads do time corrente, podem ser as seguintes:

  Ação 1 : Lançar uma nova tarefa amarrada (tarefa criada, ainda não em execução);

  Ação 2 : Retomar a execução de uma tarefa amarrada pronta que havia sido suspensa por requisitos de escalonamento ou necessidade de sincronização e que esteja com os requisitos de sincronização satisfeitos;

  Ação 3 : Lançar uma nova tarefa não amarrada;

  Ação 4 : Retomar a execução de uma tarefa não amarrada pronta que havia sido suspensa por requisitos de escalonamento ou necessidade de sincronização e que esteja com os requisitos de sincronização satisfeitos.

- A decisão de qual ação de escalonamento a ser executada é definida considerando:

  Decisão 1 : Nenhuma tarefa que possua dependências não atendidas pode ser selecionada;

[2] Grafo dirigido de tarefas, com a sigla em inglês para *Direct Acyclic Graph*.

Decisão 2 : Nenhuma tarefa sujeita a regime de exclusão mútua com outra tarefa em execução pode ser selecionada;

Decisão 3 : A **Ação 1** somente é permitida caso a lista de tarefas suspensas mas prontas para executar (com restrições de sincronização satisfeitas) não possua nenhuma tarefa amarrada;

Decisão 4 : Se uma tarefa é criada tendo a expressão associada à cláusula `if` resultando em *falso*, a tarefa é executada imediatamente, suspendendo, durante esta execução, a tarefa geradora da nova tarefa.

As consequências do escalonamento devem ser de conhecimento do programador para que ele possa compreender o impacto delas no seu programa. Fica claro que tarefas não amarradas podem migrar entre threads. Assim, durante sua execução, caso a tarefa acesse a região de memória privada ao thread, esta região pode ser alterada. Outro aspecto é em relação a situações de impasse. O uso da diretiva `critical` de forma aninhada não é permitido pois implica em uma iminente situação de impasse. Já o uso de mutex pela biblioteca de serviços permite o uso de mutex recursivo, mas neste caso as tarefas devem ser amarradas e ainda assim há o risco de impasse em função da relação de dependência de sincronização entre tarefas não ser observada diretamente pelo escalonamento – a prevenção de situações de impasse, nestes casos, pode fazer uso da invocação explícita de ações de escalonamento (`taskyield`).

2.6 Suíte de Benchmarks Cowichan

Os problemas Cowichan (WILSON; IRVIN, 1995) consistem em um conjunto de programas concebido para avaliar a usabilidade de modelos de programação paralela em termos da capacidade de expressão de sua API.

Além de serem bastante conhecidos, os problemas Cowichan respondem satisfatoriamente ao apresentado em (KESTOR et al., 2011) no que diz respeito às características desejáveis para um benchmark para TM: (*i*) inclusão de uma implementação baseada em *lock* junto com a implementação com transações; (*ii*) aplicações devem ter uma boa escalabilidade; (*iii*) aplicações devem representar problemas de mundo real; (*iv*) é necessário incluir uma variedade de comportamentos de implementação de TM (chamadas internas, aninhamentos etc); (*v*) as aplicações devem tratar de problemas atuais de pesquisa; (*vi*) e por fim, é ideal que o *benchmark* completo seja útil para avaliação de sistemas STM.

Os problemas Cowichan foram desenvolvidos para avaliar o poder de expressão das ferramentas de programação paralela. Individualmente consiste de aplicações simples. Essas aplicações, quando combinadas, cobrem vários domínios de cunho

científico. O diferencial desta suíte de benchmarks está na combinação das implementações, onde a saída de uma se torna a entrada de outra. Esta característica permite definir o quanto determinada ferramenta suporta modularização e reuso, além de retratar de forma mais fiel o funcionamento de um software, o qual é composto por vários módulos com diferentes layouts de dados e diferentes estratégias de paralelização.

O cowichan é composto por diferentes algoritmos, chamados de *toys*. A suíte possui catorze algoritmos no total. Neste trabalho foram utilizados seis algoritmos em que foi identificada a aplicabilidade do modelo de memória transacional, que são: hull: Convex Hull, norm: Point Location Normalization, outer: Outer Product, sor: Successive Over-Relaxation, thresh: Histogram Thresholding e vecdiff: Vector Difference. São apresentados a seguir (WILSON; IRVIN, 1995) (WILSON, 2010):

- **hull: *Convex Hull***. Dado um conjunto de (x,y) pontos, esse módulo encontra os que se localizam no fecho convexo, os remove, e continua a aplicação da mesma operação nos remanescentes, até que não hajam mais pontos. A saída é uma lista de pontos na ordem que estes foram removidos.

- **norm: *Point Location Normalization***. Esse módulo normaliza as coordenadas dos pontos para que todos estes se encontrem dentro da unidade quadrada $[0..1] \times [0..1]$. Se as variáveis $xmin$ e $xmax$ são as coordenadas máximas e mínimas de x no vetor de entrada, então a equação de normalização é $x_i = (x_i - x_{min})/(x_{max} - x_{min})$. As coordenadas y são normalizadas da mesma maneira.

- **outer: *Outer Product***. Esse módulo transforma um vetor contendo posições de pontos em uma matriz $N \times N$ densa, simétrica e com dominância diagonal, realizando o cálculo das distâncias entre os pares de pontos. Também constrói um vetor de valores reais, os quais são a distância de cada ponto em relação a sua origem. Cada elemento da matriz M_{ij} onde $i \neq j$, gera d_{ij}, a distância Euclidiana entre os pontos i e j. Os valores diagonais M_{ii} são definidos como sendo N vezes o valor máximo da diagonal de fora para garantir que a matriz possui dominância diagonal. O valor do elemento vi do vetor é definido como a distância entre o ponto i e sua origem, dado pela equação $\sqrt{x_{i2} + y_{i2}}$.

- **sor: *Successive Over-Relaxation***. Esse módulo resolve uma equação matricial $AX = V$, para uma matriz A, densa, simétrica, com dominância diagonal e um vetor arbitrário (não nulo) V, utilizando o método de sobre-relaxamento sucessivo.

- **thresh: *Histogram Thresholding***. Esse módulo realiza a limiarização por equilíbrio do histograma em uma imagem. Dado uma imagem inteira I e uma

porcentagem alvo p, esta constrói uma imagem binária B assim Bi,j é definida se não mais que p por cento dos pixels em I possuem mais brilho que I_{ij}. A ideia geral é a de que um histograma de uma imagem deveria ter dois picos, um centralizado ao redor da intensidade média do plano superior e o outro centralizado ao redor da intensidade média do plano inferior. É realizada a tentativa de se definir o limiar entre esses dois picos no histograma e selecionar os pixels acima desse limiar.

- **vecdiff: *Vector Difference***. Esse módulo encontra o máximo absoluto entre dois vetores com número reais, elemento por elemento.

A escolha da utilização dos problemas Cowichan se dá exatamente por cobrir aplicações de mundo real e apresentar problemas atuais a serem estudados. Deseja-se assim, demonstrar uma comparação entre os códigos originais e sua extensão ao suporte de TM, realizando sua implementação para uma definição e análise completa do caso de estudo. Os parâmetros de análise escolhidos se relacionam diretamente ao programador, por exemplo: número de linhas de código, definição de áreas críticas, e manuseio de estruturas complexas de concorrência e suas consequências.

2.7 GCC-TM

O suporte para TM no GNU C Compiler (GCC) (Free Software Foundation, 2012) (GNU TM Library, 2011) foi adicionado na versão 4.7. O objetivo da inclusão do mecanismo transacional é possibilitar que programadores possam desenvolver novas aplicações utilizando tal mecanismo, habilitar a utilização de TM em códigos existentes e receber contribuições da comunidade para otimizar e expandir o suporte. O suporte é composto de duas partes (HONORIO, 2018):

- **Anotações transacionais**. O GCC oferece mecanismo para anotar blocos de código e atributos de função para instrumentar seletivamente acessos em uma função. Acessos à memória dentro de blocos `__transaction_atomic` são instrumentados com chamadas à biblioteca em tempo de execução que implementa barreiras transacionais usadas para detectar conflitos. Todo bloco `__transaction_atomic` é transformado em dois caminhos: um com as barreiras transacionais que serão usadas por STMs e outro, sem barreiras, que pode ser usado com HTM ou uma estratégia de sincronização alternativa, como por exemplo, usando um lock global. A especificação suporta duas anotações principais de funções: `transaction_safe` e `transaction_pure` (Intel, 2012).
  - **`transaction_safe`**: consiste de uma anotação que define uma função ser segura para uma transação. Em transações atômicas, somente são invocadas funções seguras. Se a função invocada faz parte do mesmo módulo

de compilação da transação, ou seja, faz parte do mesmo arquivo fonte, o compilador pode inferir automaticamente se essa função é segura. Caso contrário, a função precisa ser anotada com o atributo `transaction_safe`. Se uma função com essa anotação possui operações inseguras, um erro de compilação é gerado. Para toda função anotada com esse atributo, o compilador gera duas versões: uma instrumentada, com barreiras de leitura e escrita necessárias para, caso seja necessário, permitir o cancelamento da transação; e uma não instrumentada, que pode ser usada fora de transações. Se uma função for anotada com esse atributo e o compilador não encontrar versão instrumentada, um erro de compilação é gerado.

  - **`transaction_pure`**: funções anotadas com esse atributo dizem ao compilador que elas não causam nenhum tipo de efeito colateral. Portanto, todos os acessos à memória dentro dessas funções não são instrumentados com barreiras.

- **Bibliotecas em tempo de execução**. A biblioteca transacional no GCC utilizada é chamada de libITM (GNU TM Library, 2011) (Intel, 2012), uma implementação que segue as linhas da ABI da Intel. Como sua documentação é semelhante à da ABI da Intel, a GNU resolveu utilizar a documentação da ABI da Intel como referência para a libITM, tendo como foco apenas as diferenças entre elas. As mudanças abrangem restrições adicionadas, funções removidas, adicionadas, alteradas ou ainda não implementadas e o modelo de memória, que deve ser definido na ABI da libITM, diferentemente da ABI da Intel, que não exige um modelo de memória. As outras duas categorias, declarações e anotações, são parte do front-end do suporte transacional do GCC, onde o código fonte é compilado e o arquivo objeto é gerado.

2.8 TinySTM

A TinySTM (FELBER; FETZER; RIEGEL, 2008) é uma implementação conhecida de Memória Transacional em Software para as linguagens C e C++, que utiliza uma abordagem de versionamento global (contador compartilhado como *clock*) para controlar os conflitos entre transações e mutexes para proteger os locais da memória compartilhada. Duas estratégias para acessos à memória foram implementadas na TinySTM: *write-through*, que utiliza versionament adiantado, e *write-back*, que utiliza versionamento atrasado. A primeira estratégia permite que as transações escrevam diretamente na memória e revertam suas atualizações no caso de precisarem ser abortadas, enquanto a segunda atrasa as atualizações de memória até o tempo de realização do *commit*.

A TinySTM usa um array compartilhado de mutexes para gerenciar acessos si-

multâneos à memória. Cada mutex cobre uma parte do espaço de endereçamento e os endereços são mapeados para mutexes com base em uma função hash. Como as transações de escrita devem verificar se todos os endereços lidos ainda são válidos – ou seja, não estão bloqueados por outra transação e ainda têm o mesmo número de versão – no momento da realização do *commit*, dependendo do número de operações de leitura e escrita, esta verificação pode ter um custo alto. Para resolver esse problema, os desenvolvedores propõem uma estratégia de bloqueio hierárquico. Nessa estratégia, as folhas da árvore – último nível na hierarquia – correspondem a elementos do array compartilhado de mutexes, enquanto os níveis superiores agregam informações sobre níveis inferiores. Assim, quando não há nenhum mutex adquirido para qualquer elemento de uma determinada sub-árvore, não é necessário validar seus elementos (CASTRO et al., 2010).

Os três parâmetros mais importantes da TinySTM são: a função hash que mapeia um local de memória para um mutex, o número de entradas no array de mutexes e o tamanho do array usado para o mutex hierárquico. Por isso, os desenvolvedores propõem uma estratégia de ajuste dinâmico para ajustar automaticamente esses parâmetros de acordo com a carga de trabalho e mostrar que isso resulta em importantes ganhos de desempenho (FELBER; FETZER; RIEGEL, 2008).

2.9 Considerações

A programação concorrente é de extrema relevância em um contexto em que arquiteturas de processamento paralelo, fomentadas pelo advento e popularização das arquiteturas multicore, se tornam onipresentes. No entanto, ferramentas para programação concorrente, em particular, programação em arquiteturas multiprocessadas, foco deste trabalho, ainda não dispõem de facilidades suficientes para suprir as necessidades em ambiente de produção de software. A realidade, portanto, indica que a programação concorrente não é mais exclusiva de programadores especializados, como aqueles voltados ao desenvolvimento de aplicações para o processamento de alto desempenho, já acostumados a utilizar os mecanismos tradicionais, ilustrados neste capítulo pelos mutexes. Toda uma nova geração de programadores, voltados a aplicações com fins comerciais, administrativos ou de entretenimento, deve ser também habilitada a explorar tais recursos de processamento. As Memórias Transacionais prometem ser uma alternativa interessante, sendo motivada por fornecer uma abstração de programação que permite obter programas mais robustos e com a propriedade de composabilidade.

Memória Transacional promete aliviar muitos dos desafios da programação paralela envolvendo mutexes. Ela libera o programador de detalhes de sincronização de baixo nível, como lidar com vários mutexes e protocolos de sincronização comple-

xos. Ela também elimina a possibilidade de postergação indefinida devido à ordem incorreta dos bloqueios. Como consequência, um mecanismo de *runtime*, particularmente em ambientes onde a Memória Transacional é oferecida em software, assume diversas responsabilidades sobre o controle da execução, e eventualmente reexecução, de transações.

Segundo (PANKRATIUS; ADL-TABATABAI, 2014), a Memória Transacional também tem armadilhas. Ainda é possível ter condições de corrida, assim como transações de granularidade grossa podem prejudicar a escalabilidade e o desempenho. Por exemplo, os programadores podem tentar otimizar seu código baseado em transações reduzindo o tamanho dos blocos atômicos, movendo o código para fora dos blocos atômicos ou quebrando os blocos atômicos em blocos menores. Essas transformações podem introduzir inadvertidamente condições de corrida em que uma variável é acessada concorrentemente dentro e fora de uma transação.

Dentre as limitações ao uso de Memória Transacional em Software, identifica-se uma questão relacionada à solução recorrente de oferecer este modelo de sincronização na forma de serviços de uma biblioteca. As funcionalidades de uma biblioteca, de certa forma, estendem a capacidade de expressão de uma linguagem de programação. No entanto, possuem mecanismos de operação definidos fora do contexto desta linguagem e, portanto, não tratadas pelo compilador, tão pouco consideradas na semântica da própria linguagem. Seu uso, portanto, é sujeito a erros de programação não tratados pelo compilador da linguagem. De forma semelhante, deve existir uma compatibilização entre os dados manipulados no programa de aplicação com aqueles suportados pela biblioteca. Neste caso, mais uma vez, o compilador se mostra inoperante na verificação destas conversões de tipo.

Pesquisas sobre a adição de suporte à Memória Transacional às linguagens de programação descobriram várias compensações no projeto da linguagem e no tipo de recursos relacionados à Memória Transacional que podem ser fornecidos ao programador. Mais estudos empíricos e experimentos com programas paralelos do mundo real são necessários para ajudar a avaliar as compensações certas a serem feitas e quando é apropriado usar a Memória Transacional (PANKRATIUS; ADL-TABATABAI; OTTO, 2009). Seus proponentes argumentam que a combinação de controle automático de concorrência de granularidade fina, atomicidade de falha automática e potencial reduzido para situações de interblocagem, fazem com que a Memória Transacional seja superior aos mutexes no suporte a programação modular. Os defensores também argumentam que, embora a Memória Transacional não seja uma solução mágica para a programação paralela, é um passo na direção de fornecer um mecanismo de controle de concorrência muito mais robusto e produtivo em comparação com a utilização de mutexes (PANKRATIUS; ADL-TABATABAI, 2014).

Na literatura também encontram-se ferramentas de programação que oferecem

soluções para programação com suporte ao modelo de Memória Transacional em programas multithread. Duas destas ferramentas foram discutidas neste capítulo, TinySTM e GCC-TM. Enquanto a primeira tem origem o meio acadêmico, a segunda está inserida em um contexto de busca de padronização. Ainda assim, ambas são amplamente acessíveis e, em programas C/C++, podem ser combinadas com quaisquer ferramentas de programação multithread.

No Capítulo 3 serão consideradas ferramentas de programação que exploram Memória Transacional no contexto da programação multithreaded.

3 ESTADO DA ARTE E TRABALHOS RELACIONADOS

Este trabalho é desenvolvido no contexto de facilitar a programação multithreaded com o uso de MT em ferramentas de programação, especificamente em OpenMP, de forma integrada ao modelo de programação oferecido por esta ferramenta e não como um recurso transversal ao modelo de programação por ela oferecido. O estado da arte, por consequência, aborda os trabalhos na literatura que propuseram extensões semelhantes. O capítulo inicia com a Seção 3.1 caracterizando o contexto da programação multithreaded, posicionando a interface proposta por OpenMP. A Seção 3.2 relaciona trabalhos que discutem a integração de TM com ferramentas de programação multithreaded, outras que não OpenMP, e, então, a Seção 3.3, trabalhos que apresentam propostas de integração de TM com OpenMP são apresentados. O fechamento do capítulo, Seção 3.4, se dá com a apresentação de um posicionamento sobre o estudo realizado.

3.1 Modernas Interfaces para Programação Multithread

Ferramentas modernas para programação multithreaded (CAVALHEIRO; DU BOIS, 2014) oferecem abstrações suficientes para que a concorrência das aplicações seja descrita em termos de um programa concorrente e executada eficientemente em um hardware multiprocessado. A estratégia adotada na implementação de várias destas ferramentas é baseada na implementação de um modelo de threads $n \times m$ (SILBERSCHATZ; GALVIN; GAGNE, 2008), onde n tarefas concorrentes são geradas pelo programa em execução e então mapeadas, por um mecanismo de escalonamento, sobre m unidades de execução paralela. Ferramentas disponíveis no mercado, como OpenMP (DAGUM; MENON, 1998) (OPENMP, 2018), Cilk Plus (BLUMOFE et al., 1995) (INTEL, 2018) e TBB (REINDERS, 2007) (OpenCilk, 2020), são exemplos de ferramentas implementadas no modelo de threads $n \times m$.

À medida que os programas multithread aumentam em tamanho e em complexidade, atendendo a crescente demanda da sociedade por software paralelo, abstrações mais avançadas são necessárias para reduzir o impacto da complexidade que

a programação concorrente introduz pelo uso frequente da sincronização em sistemas de software em larga escala (WONG et al., 2014). Em consequência, observa-se uma evolução na interface de programação concorrente oferecida por ferramentas de uso consolidado, como OpenMP e o próprio C++ (STROUSTRUP, 2013), e também o surgimento de novas ferramentas, como TBB e CnC (BUDIMLIć et al., 2010), com novas abordagens.

Neste contexto, OpenMP é considerada uma ferramenta de programação consolidada pelo seu tempo de uso em aplicações comerciais, pelo esforço de um grande número de empresas na sua padronização e pela grande quantidade de aplicações já desenvolvidas. Já na versão 4.5, a interface de programação de OpenMP evoluiu incluindo novos recursos de programação. No entanto, no que diz respeito à inclusão de novas abstrações na interface de programação para controle de acesso a dados compartilhados, novas adições não foram propostas, oferecendo, esta ferramenta, os mesmos recurso de sincronização mecanismos baseados em seções críticas de sua apresentação inicial e na definição de modos de acesso a variáveis pela parametrização de tarefas quando da realização de uma operação de criação de tarefas.

3.2 Alternativas para Integração de Memória Transacional

Esta seção relaciona trabalhos na literatura que discutem a integração de Memória Transacional com ferramentas de programação multithread.

O trabalho (SHRI; RESHMA, 2019) apresenta uma discussão sobre as semelhanças entre a Memória Transacional e a coleta de lixo, desde os problemas que eles resolvem, à maneira como os resolvem, e como a prática de programação inadequada pode anular suas vantagens. Segundo os autores, a conclusão mais importante que surge do trabalho é que coleta de lixo e TM dependem de aproximações simples e geralmente boas o suficiente (a saber, acessibilidade e conflitos de memória) que estão sujeitas a problemas de compartilhamento falso. Esse fato pode informar como os programadores são instruídos a usar a TM (e coleta de lixo) de forma eficaz e pode orientar a pesquisa para reduzir as aproximações.

O trabalho (MICULAN; PERESSOTTI, 2020) apresenta o Open Transactional Memory (OTM), uma abstração de programação que oferece suporte a interações seguras e orientadas por dados entre transações de memória composáveis. Isso é alcançado relaxando o isolamento entre as transações, ainda garantindo a atomicidade. Esse modelo permite interações fracamente acopladas, uma vez que a fusão de transações é conduzida apenas por acessos a dados compartilhados, sem a necessidade de especificar os participantes com antecedência. Os autores propõem um modelo de programação para interações seguras e orientadas por dados (data-driven) entre transações de memória, fornecendo uma breve visão geral da interface

para transações abertas para Haskell. Segundo os autores, OTM pode ser implementado em qualquer linguagem de programação, desde que se tenha alguns meios para proibir efeitos irreversíveis dentro das transações. Haskell foi a linguagem escolhida pois seu sistema de tipagem permite implementar essa restrição com bastante facilidade. Este modelo separa transações isoladas de não isoladas, ainda garantindo atomicidade; o último pode interagir acessando as variáveis compartilhadas. A consistência é garantida pela fusão transparente das transações de interação em tempo de execução. Os autores ainda fornecem uma semântica formal para OTM e provam que este modelo satisfaz o importante critério de opacidade. O critério de correção de opacidade é uma extensão da propriedade de serialização clássica para bancos de dados com o requisito adicional de que até mesmo transações não confirmadas devem acessar estados consistentes.

Em (MEDIĆ et al., 2020) é apresentado um framework para descrever STMs. Em particular, quando uma transação é abortada, são descartadas todas as atualizações realizadas e necessário trazer o sistema de volta ao estado anterior à execução da transação. Para realizar o comportamento acima, os autores implementaram um operador de rollback seguindo a abordagem dada em (LANESE et al., 2011). Uma transação pode acessar uma variável compartilhada em modo de leitura ou escrita. Diante disso, diferentes políticas podem ser utilizadas para regular as transações que estão acessando a mesma posição de memória. De acordo com a política implementada, algumas transações serão bem-sucedidas e outras abortadas. O framework é capaz de modelar duas políticas diferentes para a execução das transações concorrentes: priorizando operações de escritas ou de leituras. A intenção dos autores é começar a partir de um cálculo simples e, em seguida, adicionar de forma modular: transações aninhadas, estruturas de dados (por exemplo, estruturas C) e políticas de escalonamento mais complexas. As transações aninhadas exigirão registrar para cada transação uma lista de suas transações filhas. Esses filhos herdam o acesso da transação pai, podendo ser aninhamento aberto ou fechado. O objetivo final dos autores foi, então, provar que a estrutura modular satisfaz a propriedade de opacidade. Tal critério é o mesmo utilizado em (MICULAN; PERESSOTTI, 2020).

Nos trabalhos (SWALENS; DE KOSTER; DE MEUTER, 2018) e (SWALENS; DE KOSTER; DE MEUTER, 2017), três esquemas de sincronização, futuros, atores e transações, são estudados e é analisado o resultado de usos combinados em pares: futuros-transacionais e atores-transacionais. A contribuição deste artigo, segundo seus autores, é sua integração em uma única estrutura que une todos os três modelos de maneira coerente. No trabalho é mostrado que uma combinação ingênua desses modelos invalida as garantias que eles normalmente fornecem, quebrando assim as suposições de desenvolvedores. Por isso, é apresentada a Chocola: uma estrutura unificada de futuros, atores e transações que mantém as garantias de to-

dos os modelos, sempre que possível, mesmo quando eles são combinados. Chocola é construída a partir de Clojure, uma variante de Lisp, executada sobre a Java Virtual Machine. Portanto, a sintaxe de Chocola e suas funções integradas são iguais às de Clojure. Clojure apoia futuros e transações, que Chocola reutiliza e estende para apoiar atores. Seu desempenho e expressividade são avaliados usando três aplicações do conjunto de benchmarks STAMP (Chi Cao Minh et al., 2008). Os testes demonstraram que misturar vários modelos melhora seu desempenho, introduzindo um paralelismo mais refinado. Essas transformações não alteram fundamentalmente o design do programa e, portanto, exigem apenas um esforço relativamente pequeno do desenvolvedor. Portanto, usando o Chocola, os desenvolvedores podem escolher e combinar modelos de concorrência livremente em seus programas.

Em (ZARDOSHTI et al., 2019) é apresentada uma abordagem alternativa para oferecer suporte à TM em C++, proibindo o auto-*abort* explícito e introduzindo um mecanismo baseado em executor para a execução de transações. Tal abordagem torna mais fácil para os desenvolvedores colocar o código em funcionamento com a TM. Segundo os autores, a proposta também deve ser atraente para os desenvolvedores de compilador, uma vez que permite um espectro de níveis de suporte para TM, com desempenho variável e dependência variável de suporte de TM em hardware para fornecer escalabilidade. O projeto apresentado para transações no estilo executor em C++ não introduz novos requisitos para programadores. Em vez de marcar as transações como regiões léxicas, como um bloco de instruções, os programadores envolvem as transações em lambdas e as passam para um executor. Na avaliação de desempenho realizada é mostrado que a abordagem introduz uma sobrecarga de instrumentação, variando de 2% a 40% em comparação com código não instrumentado e apresentou uma escalabilidade equivalente a oferecida por GCC-TM.

Em (BUSCH et al., 2017) o contexto da pesquisa é na aplicação de TM em ambientes com memória distribuída, em uma interface de programação orientada a objetos. O tema de investigação são algoritmos de escalonamento, onde transações que residem em nós de um grafo de comunicação operam em objetos móveis compartilhados. O esquema de escalonamento deve considerar a mobilidade dos objetos para permitir a execução das transações. Uma transação solicita os objetos de que precisa, executa uma vez que esses objetos estão disponíveis e, em seguida, possivelmente encaminha esses objetos para outras transações em espera. Sendo a principal contribuição deste artigo o escalonamento, em conclusão, são apresentados os limites de desempenho, superior e inferior, para diferentes organizações de comunicação.

Em (TABASSUM; MEENU, 2020) é apresentada uma revisão de literatura sobre uso de STM em Python. Uma das principais características de Python é a clareza de

seu código, facilitando a compreensão dos programas (DIERBACH, 2014). Sua primeira versão data do início da década de 1990 e é, atualmente, uma das linguagens de programação mais populares (SRINATH, 2017). O artigo apresenta um conjunto de trabalhos que desenvolvem o uso de TM sobre esta linguagem de programação. São apresentados tanto trabalhos que estendem a API de Python para utilizar esta TM, como trabalhos que associam o uso de bibliotecas STM, TinySTM, notadamente, sobre Python.

O trabalho de (BONNICHSEN; PODOBAS, 2015) não propõe a extensão da interface de programação de alguma linguagem de programação. Seu objetivo é explorar como a Memória Transacional em Hardware pode ser utilizada para reduzir o bloqueio nas diretivas de sincronização OpenMP. São apresentados métodos baseados em HTM para executar as diretivas OpenMP `barrier`, `critical`, e `taskwait`, sem bloqueio. Os testes realizados no trabalho mostram uma melhoria no desempenho em relação às abordagens de bloqueio tradicionais e também se mostra melhor do que outras abordagens HTM em seções críticas.

Os trabalhos citados nesta seção, indicam uma regularidade da abordagem da associação do modelo de Memória Transacional com os recursos de programação e/ou execução oferecido por ferramentas de programação. Neste tipo de abordagem, a questão central é como oferecer a semântica de manipulação de dados em Memória Transacional de forma coerente com a proposta da própria ferramenta, considerando sua interface de programação ou ainda seu modelo de execução. Deste tipo de abordagem, a Seção 3.3 apresenta um recorte da literatura considerando trabalhos que buscam integrar TM com OpenMP.

3.3 Integração de Memória Transacional e OpenMP

Esta seção aborda os trabalhos na literatura que propuseram extensões da interface de programação de ferramentas de programação multithreaded para contemplar o uso de Memória Transacional.

No estudo realizado por (HEUVELINE et al., 2013) foi avaliada a aplicabilidade da Memória Transacional para o método de Gradientes Conjugados (GC), um solucionador para sistemas lineares de equações frequentemente usado em muitos campos de aplicação, especialmente na área de mecânica estrutural e dinâmica de fluidos computacionais. Observa-se que este trabalho não propõe uma interface para OpenMP suportando Memória Transacional. No entanto ele é apresentado por representar trabalhos que comparam o desempenho de aplicações desenvolvidas com e sem suporte à Memória Transacional, utilizando uma versão desenvolvida em OpenMP para comparação de desempenho. No trabalho, foi realizada uma comparação entre três implementações similares do Método *Conjugate Gradients*. Uma que

usa o OpenMP, uma que usa Pthreads sem Memória Transacional e uma que usa Pthreads com construções de Memória Transacional, construções estas oferecidas por TinySTM. Primeiro implementaram o algoritmo GC usando a linguagem de programação C e OpenMP. Posteriormente, este código foi transformado em uma variação similar de Pthreads e, por último, esta versão foi modificada usando comandos de Memória Transacional. Enquanto as duas últimas ferramentas exploram conceitos tradicionais, baseados em exclusão mútua, para gerenciar paralelização e sincronização, os mecanismos baseados em Memória Transacional de Software foram projetados para lidar apenas com o último. No programa OpenMP, a paralelização é conseguida inserindo `#pragma omp for` no topo de cada laço `for`. Devido às barreiras implícitas, não foi preciso se preocupar com as dependências de dados entre os vários laços `for`. Segundo os autores, os resultados mostraram que é muito importante reduzir o tempo de espera nas barreiras, a fim de melhorar o tempo de execução dos programas. Na maioria dos casos, o OpenMP é a abordagem mais rápida nos testes realizados.

Em (MILOVANOVIĆ et al., 2007) e (MILOVANOVIĆ et al., 2008) é apresentado o framework Nebelung para manipulação de Memória Transacional e seu uso combinado com OpenMP. Nebelung provê um mecanismo de memória transacional em software. Sua implementação conta com um núcleo de execução multiprocessado e de um compilador, o Mercurium. Ambos trabalhos discutem questões de projeto de linguagem na associação de OpenMP com Memórias Transacionais. Eles propõem extensões OpenMP para Memória Transacional, como por exemplo, a mostrada na Figura 7.

```
1  #pragma omp transaction [exclude (list) | only (list)]
2        bloco estruturado
```

Figura 7 – Sintaxe da diretiva `transaction` em Nebelung, proposta por (MILOVANOVIĆ et al., 2007) e (MILOVANOVIĆ et al., 2008)

A extensão é um `pragma` para delimitar a sequência de instruções que compõem uma transação, identificada pela nova diretiva **`transaction`**. Com esta extensão, o programador pode escrever programas OpenMP padrão, sem a necessidade de incluir mecanismos clássicos de sincronização baseados em exclusão mútua, como os oferecidos pelas diretivas de `atomic` e `critical`. O `bloco estruturado` define a sequência de instruções que devem ser executadas como uma transação.

Na extensão proposta, a cláusula `exclude`, que é opcional, pode ser usada para especificar a lista de variáveis para as quais não é necessário verificar conflitos. Isso significa que a biblioteca de Memória Transacional de Software não precisa acompanhá-los nos conjuntos de leitura e escrita. Pelo contrário, se o programador usa a cláusula opcional `only`, ele especifica explicitamente a lista de variáveis

que precisam ser rastreadas, neste caso, o versionamento de dados para todas as escritas especulativas se aplica a todas as variáveis compartilhadas e privadas no caso de a transação precisar reverter (*roll-back*).

No framework, o módulo Nebelung consiste em uma biblioteca oferecendo primitivas para manipulação de Memória Transacional em Software. Este módulo implementa um núcleo de execução multithreaded para detecção adiantada de conflitos assíncronos. A ideia é que as transações não desperdicem tempo de processamento dos threads de execução, aqueles fornecidos pelo núcleo de execução de OpenMP, para detecção adiantada de conflitos. As transações, ao serem lançadas, simplesmente enviam uma mensagem para o thread de detecção de conflito (*conflict detection thread* – CDT), thread adicional separado para detecção adiantada de conflito assíncrono, que notifica de forma assíncrona a transação se surgir algum conflito. A biblioteca de Nebelung realiza detecção adiantada de conflito e gerenciamento preguiçoso de versionamento. Segundo os autores, os resultados com o thread de detecção de conflito são muito melhores por dois motivos: (*i*) o thread de detecção de conflito executa a detecção adiantada de conflito; e (*ii*). não há sobrecarga para detecção adiantada de conflito nas transações em execução porque é realizada no thread dedicado. A única sobrecarga para este tipo de sistema é sobrecarga para enviar a mensagem para o thread de detecção de conflito.

O trabalho (BAEK et al., 2007) apresenta o OpenTM, que estende OpenMP com diretivas para expressar sincronização não bloqueante e paralelização especulativa baseada em transações de memória. O OpenTM foi projetado como uma extensão do OpenMP para suportar sincronização não bloqueante e paralelização especulativa usando técnicas transacionais. Por isso, OpenTM herda o modelo de execução OpenMP, semântica de memória, sintaxe de linguagem e construções de tempo de execução. Qualquer programa OpenMP é um programa OpenTM legítimo. Código não transacional, paralelo ou sequencial, se comporta exatamente como descrito na especificação OpenMP. O modelo transacional para OpenTM é baseado em três conceitos chave. **Isolamento forte**: transações de memória são atômicas e isoladas em relação a outras transações e acessos não-transacionais. **Transações implícitas**: o programador define os limites das transações em blocos paralelos, sem anotações adicionais para acessos a dados compartilhados. Neste caso, todas as operações de memória são executadas implicitamente dentro da transação para garantir atomicidade. **Transações virtualizadas**: o sistema de Memória Transacional deve garantir a execução correta mesmo quando as transações excederem o quantum de tempo, excederem a capacidade de caches de hardware ou memória física, ou incluirem uma grande quantidade de níveis de aninhamento.

A interface de programação proposta por OpenTM é composta por três diretivas que estendem as funcionalidades de OpenMP com recursos para manipulação de

Memória Transacional: **`transaction`**, **`transfor`** e **`transections`**/**`transection`**. Estas diretivas são discutidas na sequência.

Transação: O modelo básico de manipulação de Memória Transacional é provido pela diretiva **`transaction`**. Esta diretiva especifica os limites de um bloco estruturado onde ocorrem transações fortemente isoladas. Em um programa OpenTM, a diretiva substitui mecanismos clássicos de exclusão mútua baseados em mutex, seções críticas e operações atômicas. A estrutura de uso desta diretiva é dada conforme mostrado na Figura 8.

```
1  #pragma omp transaction [clausula [[,] clausula] ...]
2      bloco estruturado
```

Figura 8 – Sintaxe da diretiva `transaction`, proposta por (BAEK et al., 2007)

Onde `cláusula` é uma das seguintes: `ordered`, `nesting(open|closed)`. `Ordered` é usada para especificar uma ordem de confirmação sequencial entre as execuções desta transação por threads diferentes, sendo útil para a paralelização especulativa do código sequencial. Se não especificada, gera transações não ordenadas e serializáveis. Transações não ordenadas são úteis para sincronização sem bloqueio em código paralelo. Durante a execução das transações, o sistema de Memória Transacional detecta acessos conflitantes a variáveis compartilhadas para garantir atomicidade e isolamento. Em um conflito, o sistema anula e executa novamente transações com base no esquema de pedidos e em uma política de gerenciamento de contenção. A cláusula `nesting` especifica o comportamento de transações aninhadas. Se `nesting` não for especificada, o OpenTM usa transações aninhadas fechadas por padrão. As transações aninhadas fechadas podem abortar devido a dependências sem que seus pais abortem. As atualizações de memória de transações aninhadas fechadas tornam-se visíveis para outros threads somente quando a transação mais externa realiza *commit*. O código da Figura 9 exemplifica o uso da interface proposta.

No exemplo, o programador simplesmente usa a construção `transaction` para especificar que as solicitações do cliente devem ser executadas em paralelo como transações atômicas. Não há necessidade de provar que as solicitações são independentes ou usar mutexes de baixo nível para gerenciar as dependências não frequentes. Este código atinge um bom desempenho pois o sistema de Memória Transacional subjacente implementa a concorrência otimista e os conflitos entre as solicitações do cliente não são particularmente comuns.

As diretivas **`transfor`** e **`transections`**/**`transection`**, por sua vez, estendem o conceito de transação, em OpenTM, para uso associado às diretivas originais de OpenMP **`for`** e **`sections`**/**`section`**. No primeiro caso, é considerado que todas as

```
1                    void client_run(args) {
2                        for (i = 0; i < numOPS; i++) {
3                            #pragma omp transaction
4                            { /* begin transaction */
5                                switch(action) {
6                                    case MAKE_RESERVATION:
7                                        do_reservation();
8                                    case DELETE_CUSTOMER:
9                                        do_delete_customer();
10                                   ...}
11                           } /* end transaction */ }}
12                   void main() {
13                       #pragma omp parallel {
14                           client_run(args);}}
```

Figura 9 – Exemplo da Interface do OpenTM (BAEK et al., 2007)

tarefas criadas no laço resolverão conflitos ao acesso à memória compartilhada utilizando transações. De forma equivalente, transações regerão o acesso aos dados compartilhados pelas diferentes tarefas instanciadas pelas diferentes seções descritas pela diretiva.

Em (WONG et al., 2010) os autores exploram o potencial da Memória Transacional para aplicações OpenMP. Na proposta, são combinados um sistema de Memória Transacional de software, que fornece uma primitiva de transação, e uma implementação OpenMP que fornece todas as outras funcionalidades de memória compartilhada. O sistema apresentado usa o OpenMP para gerar threads e paralelizar programas. As transações indicadas pela interface de manipulação de Memórias Transacionais servem como uma alternativa à sincronização OpenMP.

No trabalho, foi utilizada a Linguagem C para implementar o *runtime system*, o qual suporta a exploração de uma variedade de cenários de Memória Transacional (Memória Transacional em Hardware, Memória Transacional em Software e Memória Transacional em Software acelerada por Hardware) em várias linguagens, incluindo C, C++ e Java. A implementação, diferente do trabalho de (BAEK et al., 2007), possui um isolamento fraco, pois os acessos a uma determinada localização compartilhada sempre ocorrem dentro de transações (uma localização transacional). Além disso, assume que as transações incluem apenas operações revogáveis sem efeitos colaterais (por exemplo, sem arquivos de Entrada/Saída – E/S).

Para sincronizar o acesso transacional a locais de memória compartilhada, o sistema de execução utiliza metadados. Uma entrada de metadados é associada com localização transacional. Esta entrada inclui um número de versão para rastrear atualizações da localização e um mutex para proteger as atualizações. Um thread pode escrever na memória somente após realizar com sucesso a operação *lock* sobre o metadado associado. Uma transação incrementa o número da versão quando libera

o mutex de metadados, o que garante que os dados não foram alterados se o número da versão não for alterado. Uma transação pode ler um número de versão de metadados e, em seguida, os dados associados, e depois verificar o número da versão para determinar se os dados estão inalterados.

Os programas definem transações para o compilador para Memória Transacional em software pela diretiva **`atomic`**, na forma descrita na Figura 10:

```
1  #pragma tm atomic [default (trans|notrans)]
2      bloco estruturado
```

Figura 10 – Sintaxe diretiva `atomic` proposta em (WONG et al., 2010)

A cláusula `default` é opcional, sendo o valor padrão `trans`. Onde `default (trans|notrans)` define o comportamento padrão da memória referenciada da região transacional. Se o usuário especificar `default (trans)` o compilador traduz referências a variáveis compartilhadas na região transacional para chamadas de tempo de execução de Memória Transacional de Software, referidas grande parte das vezes como barreiras de leitura e escrita de Memória Transacional de Software, as quais garantem a correção da execução. Se o usuário especificar `default (notrans)`, o compilador não traduz quaisquer referências de memória na região para barreiras de leitura e escrita de Memória Transacional de Software.

Outro diferencial do trabalho foi o desenvolvimento de um novo benchmark para Memória Transacional, que explora o uso de transações para aplicações de malhas (*mesh*) não estruturadas: BUSTM (*Benchmark for Unstructured-mesh Software Transactional Memory*), que fornece um código simples para explorar os benefícios de programação de transações e quaisquer implicações de desempenho, pois imita os algoritmos e o comportamento de aplicações reais de malhas não estruturadas. A eficácia da Memória Transacional para a computação científica foi avaliada com a utilização deste benchmark.

BUSTM pode emular dois cenários distintos: aplicações de Cálculo de Mecânica de Fluídos e aplicações Monte Carlo. Em ambos os casos, conflitos para as transações são pouco frequentes; No entanto, a execução correta requer sincronização de algum tipo. Nos resultados de desempenho inicial, foi verificado que a implementação da Memória Transacional de Software superou a implementação do OpenMP equivalente em regiões críticas em 10%, resultado significativo, considerando que a Memória Transacional de software possui custos indiretos relativamente elevados. Em geral, demonstra que a extensão do OpenMP inclui transações que facilitam o esforço de programação, permitindo um melhor desempenho.

Em (BIHARI et al., 2012) e (WONG et al., 2014), os autores dão continuidade ao trabalho de (WONG et al., 2010), anteriormente descrito. Nesta continuidade

são apresentados resultados usando Memória Transacional em Hardware no sistema Blue Gene/Q da IBM. Ao mostrar como este sistema de Memória Transacional pode reduzir significativamente a complexidade da programação de memória compartilhada enquanto mantém a eficiência, os autores continuam a ampliar os resultados apresentados em (WONG et al., 2010), especificando uma interface de programação para Memória Transacional em OpenMP.

São apresentados dois tipos de blocos para explorar a Memória Transacional: blocos sincronizados e blocos atômicos, ambos compondo transações OMP. Os blocos sincronizados se comportam como se todos os blocos sincronizados fossem protegidos por um único mutex recursivo global. A sintaxe de um bloco sincronizado é mostrada na Figura 11.

```
1 #pragma omp synchronized
2     bloco estruturado
```

Figura 11 – Sintaxe - bloco sincronizado (WONG et al., 2014)

Blocos atômicos, também chamados de transações atômicas, ou apenas transações, parecem executar atomicamente e não simultaneamente com qualquer bloco sincronizado (a menos que o bloco atômico seja executado dentro do bloco sincronizado). A sintaxe de um bloco atômico é mostrada na Figura 12.

```
1 #pragma omp transaction [clausula [[,] clausula] ...]
2     bloco estruturado
```

Figura 12 – Sintaxe - bloco atômico (WONG et al., 2014)

A diretiva **`transaction`** permite especificar um tratador de exceção para o bloco atômico. A cláusula seguinte à **`transaction`** diz respeito ao especificador de exceção do bloco atômico. Especifica o comportamento quando uma exceção escapa à transação ou um *cancel* atômico do OpennMP ocorre dentro da região de Memória Transacional. Estas cláusulas são:

- `noexcept`: comportamento indefinido e não é permitido; nenhum efeito colateral da transação pode ser observado.
- `commitonesc`: a transação realiza *commit* e a exceção é lançada.
- `cancelonesc`: se a exceção for segura para transações, a transação é cancelada e a exceção é lançada. Caso contrário, é um comportamento indefinido. Em ambos os casos, não podem ser observados efeitos colaterais da transação

A avaliação dos autores mostra que a Memória Transacional oferece benefícios substanciais em todos os casos. As comparações de desempenho feitas por eles

exigiram apenas a substituição das diretivas OpenMP, mantendo a capacidade de composição e os benefícios da programação modular, mantendo a eficiência. Os resultados, segundo os autores, caracterizam Memórias Transacionais como uma alternativa para simplificar a programação paralela modular em OpenMP.

O trabalho (WONG et al., 2014), ainda apresenta um survey de trabalhos anteriores suportando casos de uso, usabilidade e desempenho da Memória Transacional em aplicações do mundo real. Diversas publicações relatam experiências reunidas no esforço de paralelizar aplicações realistas usando Memória Transacional. Geralmente são baseadas no uso de implementações de software de Memória Transacional, com resultados variados em termos de desempenho. Exemplos de tais esforços incluem a triangulação de Delaunay (SCOTT et al., 2007), floresta de abrangência mínima de gráficos esparsos (KANG; BADER, 2009) algoritmo de roteamento de Lee (ANSARI et al., 2008), servidores de jogos multiplayer como QuakeTM (GAJINOV et al., 2009) e Atomic Quake (ZYULKYAROV et al., 2009) (ABDELKHALEK; BILAS, 2004) e SynQuake (LUPEI et al., 2010); ou benchmarks STMBench7 (GUERRAOUI; KAPALKA; VITEK, 2007), STAMP (MINH et al., 2008) e RMS-TM (KESTOR et al., 2009), os estudos de Rossbach e Pankratius (ROSSBACH; HOFMANN; WITCHEL, 2010a), (PANKRATIUS; ADL-TABATABAI, 2011), e os casos de uso examinados por Gottschlich e Boehm (GOTTSCHLICH; BOEHM, 2013), todos eles compostos de um número de aplicações representativas de uma variedade de domínios de aplicação.

3.4 Considerações

Uma extensa pesquisa foi realizada em ferramentas de pesquisa acadêmica, como Google Scholar e Semantic Scholar, sobre os tópicos de Memória Transacional e OpenMP, especificamente procurando trabalhos onde ambos se relacionam. Esta relação direta foi encontrada apenas nos trabalhos relacionados citados na tese. A relação entre o número de publicações por ano envolvendo os tópicos pesquisados é ilustrada na Figura 13. Este gráfico foi gerado com dados fornecidos pela ferramenta Dimensions[1], o qual informa o número de artigos com referência a um determinado conjunto de palavras chaves a partir de uma base de dados contendo populares veículos de divulgação científica. O objetivo deste estudo é compreender o espaço que a pesquisa no tema "Memórias Transacionais" ocupa no cenário atual. Foram realizadas quatro consultas, com os seguintes termos: (*i*) *Transactional Memory*; (*ii*) OpenMP; (*iii*) *Transactional Memory* OpenMP; e (*iv*) multicore. Como informação preliminar para análise do gráfico, registra-se que a primeira especificação para OpenMP foi apresentada em 1998 e que o primeiro processador multicore foi apresentado em

[1] `https://www.dimensions.ai`

2001. A ascensão do multicore na lista Top500[2] registra no ano de 2006 com uma centena de configurações possuindo processadores dual-core.

O gráfico ressalta que o tema *Transactional Memory* teve seu pico em 2003. OpenMP, por sua vez, registra um crescimento constante, principalmente a partir deste mesmo ano, em publicações científicas. Já o crescimento dos interesses de pesquisa em multicore, a partir do ano de 2007 cresce acentuadamente. No entanto, o mesmo interesse não é refletido nas buscas combinadas dos termos *Transactional Memory* e OpenMP. 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 TBB (VOSS; ASENJO; REINDERS, 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.

Na programação multithreaded, 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; ASENJO; REINDERS, 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; ASENJO; REINDERS, 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

Encontramos, neste gráfico, também subsídios para justificar o tema escolhido desta tese de doutorado. As características de Memórias Transacionais para potencializar o desenvolvimento de software concorrente robusto e capaz de composição são condizentes com o amadurecimento da área. A expectativa não é mais apenas oferecer recursos de programação para áreas científicas, mas promover o desenvolvimento de ferramental adaptado às necessidades de uma gama mais ampla de programadores e da produção de sistemas de grande porte.

As principais ferramentas consideradas nesta tese são as apresentadas em (MI-

[2] http://www.top500.org

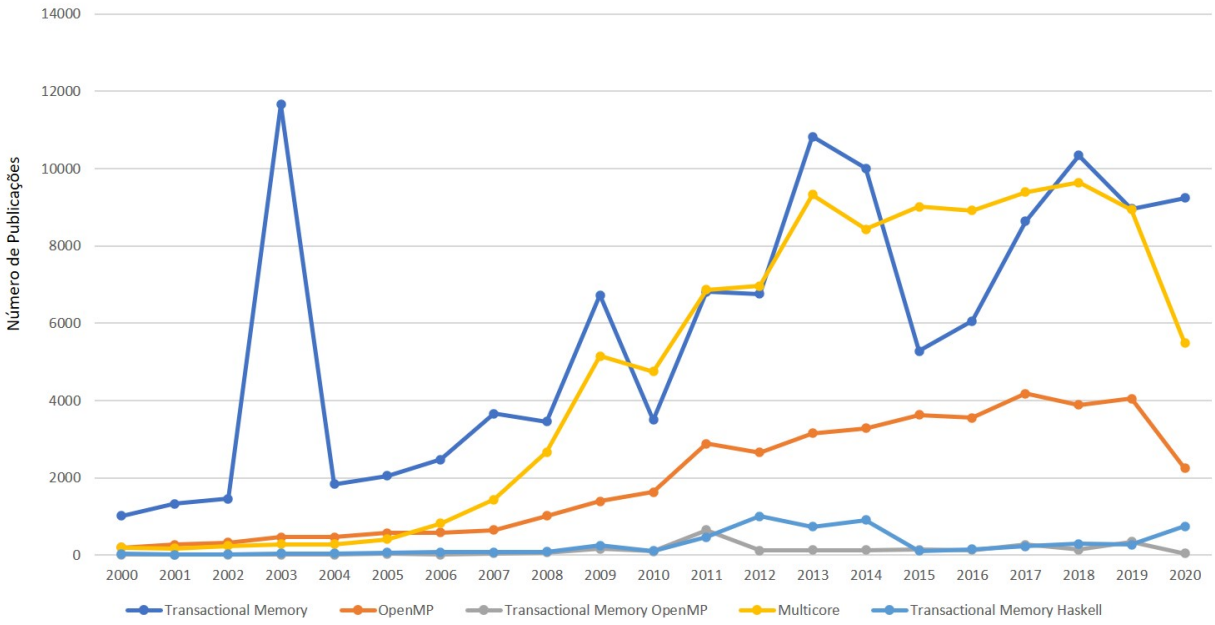


Figura 13 – Registro de Publicações dos Tópicos Estudados

LOVANOVIĆ et al., 2008), (WONG et al., 2010) e (BAEK et al., 2007), uma vez que as duas primeiras são origem de outros trabalhos considerados. Em comum, todas as propostas apresentam uma interface de manipulação de transações que estendem a oferecida por OpenMP introduzindo uma nova diretiva específica para tratar transações. Autores dos três trabalhos foram contactados e confirmaram que os projetos não estão sendo mais mantidos. Na Tabela 1 são contrastadas as principais características dos trabalhos citados neste capítulo. Abaixo, algumas destas características são discutidas.

No trabalho de (MILOVANOVIĆ et al., 2008), os autores afirmam que a diretiva `critical` do OpenMP não é uma transação pois sua implementação é baseada em mutexes. Utilizar uma diretiva `transaction` implica em uma alteração mínima do código, oferecendo um novo modelo de execução. No entanto, a dificuldade neste ponto está em introduzir um *overhead* maior monitorando todos os acessos à memória realizados no bloco, enquanto apenas os dados compartilhados deveriam ser ter seus acessos monitorados. Portanto, a utilização de `transaction` como cláusula permite identificar quais dados devem ser monitorados, oferecendo uma semantica de CRCW (*Concurrent Read, Concurrent Write*) não ainda ofertada pelos modelos `shared`, `first/lastprivate` e `reduce` disponíveis em OpenMP.

Em relação ao OpenMP, o trabalho de (WONG et al., 2010) relata o ganho de desempenho com o uso de transações no lugar de *seções críticas*. Porém, sua proposta é a de substituir a cláusula `critical`, nativa do OpenMP, oferecendo assim um recurso que sobrepõe, em função, um já existente. Já o trabalho de (MILOVANO-

VIĆ et al., 2008) apresenta uma proposta básica de como estender o OpenMP com transações em *software* e em *hardware*, e relatou alguns possíveis problemas com a definição de aninhamento de transações (aninhamento fechado ou aberto) e com seu uso em E/S.

O principal aspecto a ser considerado nos trabalhos dos grupos de Michael Wong e no Miloš Milovanović é que ambos propõem o uso de uma diretiva para transação em alternativa à diretiva para seção crítica ou então alteram a identificação de diretivas para informar que elas instanciam tarefas realizando operações transacionais. A primeira situação, pode oferecer resultados em termos de desempenho, no entanto, não oferece nenhum novo recurso de programação que de fato estenda a capacidade de representação da linguagem. No segundo caso, além de modificar a sintaxe nativa do OpenMP, não explora o fato que a sincronização pela transação se dá no dado. Alterar o nome da diretiva para indicar a necessidade de uma sincronização é contrária tanto à própria lógica de sincronização explorada em Memórias Transacionais quanto como à prática de OpenMP em permitir definir uma semântica de acesso aos dados na parametrização das tarefas.

Em relação aos problemas Cowichan, nesta tese aplicados para aferir o benefício do uso da cláusula proposta, foram encontrados outros estudos que também os utilizaram com propósitos semelhantes. Os trabalhos de (Wilson; Bal, 1996) e (WILSON, 1994) utilizaram os problemas para avaliar a linguagem de programação Orca visando melhorá-la. Além disso, ambos trabalhos buscaram descrever como os problemas Cowichan podem ser utilizados para avaliar modelos de programação. No estudo de (PAUDEL; AMARAL, 2011), o objetivo foi verificar a programabilidade da linguagem X10, identificando seus pontos fortes e fracos. O Cowichan também foi empregado para comparar a usabilidade de diferentes modelos de programação paralela (NANZ et al., 2013), onde isto é descrito utilizando Chapel, Cilk, Go e TBB. Por fim, os trabalhos de (ANVIK et al., 2005) e (FüRLINGER et al., 2018) utilizam o Cowichan para identificar a usabilidade e produtividade de novos modelos de programação que estão sendo propostos.

Tabela 1 – Comparação entre Trabalhos Relacionados

Trabalho	Versionamento	Detecção de Conflitos	Tipo de MT	Extensões para OpenMP	Aninhamento	Questões em aberto
(HEUVELINE et al., 2013)	-	-	STM	-	não	-
(MILOVANOVIĆ et al., 2007)	preguiçoso	adiantado (em um thread separado)	STM (customizável para trabalhar com sistemas STM, HTM, Híbridos)	sim (diretivas)	sim (plano)	aninhamento de transações; uso de E/S dentro de uma transação; aninhamento de transações e construções OpenMP; adição de funcionalidade adicionais em `transaction`.
(MILOVANOVIĆ et al., 2008)	preguiçoso	adiantado	STM (com suporte para HTM)	sim (diretivas)	sim	melhorar o compilador e *runtime system*; introduzir otimizações específicas de TM; melhorar agendamento para sincronização condicional; adicionar outros esquemas de gerenciamento de contenção.
(BAEK et al., 2007)	preguiçoso (HTM)	adiantado (HTM)	STM, HTM, Híbridos	sim (diretivas)	sim (aberto e fechado)	paralelismo aninhado: não é permitido código de usuário para gerar threads de trabalho extras em uma transação.
(WONG et al., 2010)	adiantado	preguiçoso (write-after-write) adiantado (read-after-write)	STM (com suporte para HTM e STM acelerada por Hardware)	sim (uma diretiva)	sim (plano)	explorar resultados de desempenho com diferentes malhas; beneficiar aplicações científicas emuladas.
(BIHARI et al., 2012)	não citado[1]	preguiçoso e adiantado (usuário escolhe)	HTM	sim (uma diretiva)	sim (plano)	inclusão da MT na especificação OpenMP
(WONG et al., 2014)	não citado[1]	não citado[1]	STM	sim (diretivas e blocos)	sim (aberto e fechado)	fornecer uma implementação usando o compilador Mercurium OpenMP da BSC ou compilador GNU para demonstrar o conceito e confirmar a capacidade de desempenho da proposta

[1] Embora os trabalhos citados não informem claramente o modelo de versionamento e de detecção de conflitos adotada, pode-se inferir que seguem os propostos em (WONG et al., 2010), uma vez que são continuidades deste.

4 MEMÓRIA TRANSACIONAL EM OPENMP

O foco deste trabalho é Memória Transacional em Software (STM, *Software Transactional Memory)*, com objetivo principal de caracterizar uma extensão à API baseada no OpenMP para linguagem C oferecendo suporte ao uso de ações transacionais. Deseja-se também aferir o benefício do uso desta extensão no desenvolvimento de programas. Para isto, foram utilizados os problemas Cowichan (WILSON; IRVIN, 1995) implementados em OpenMP.

Neste capítulo é apresentada a concepção da solução, pela cláusula proposta `transaction`. Um modelo de memória de OpenMP com transações é apresentado e, a fim de comparar a presente proposta com diferentes abordagens que propuseram o uso de TM em OpenMP, uma análise do código obtido é apresentada. O impacto de tal interface na recursão é discutido. Por fim, é apresentada a prototipação da interface proposta, na forma de uma linguagem intermediária, *Vanilla*-TM, concebida com o objetivo de permitir que a extensão proposta possa ser suportada por diferentes ferramentas de suporte a Memória Transacional em Software.

4.1 Concepção da Solução

As interfaces de programação multithread da nova geração, como OpenMP e TBB, oferecem abstrações para criar laços e tarefas de forma recursiva. Um modelo de Memória Transacional para estas ferramentas deve também contemplar estas funcionalidades. Os trabalhos (MILOVANOVIĆ et al., 2008) e (WONG et al., 2014)[1] não abordam explicitamente o uso de Memória Transacional associado à diretiva `task`, introduzida em OpenMP 3.0 no final da década de 2000, no qual a possibilidade de escrita de algoritmos concorrentes recursivos é exposta. Assim, os trabalhos encontrados como relacionados à proposta apresentada não abordam explicitamente o uso desta diretiva ou tecem considerações sobre a implicação da recursão nas suas

[1] O pesquisador Michael Wong (`https://wongmichael.com`) se mantém ativo nos esforços de padronização da concorrência na linguagem C++ e no seu suporte em GCC. O trabalho citado neste ponto refere-se ao último de uma série de suas publicações no contexto de suporte à Memória Transacional em OpenMP.

abordagens.

Em OpenMP, nas ferramentas discutidas no Capítulo 3, o uso de transações se dá com a utilização de uma nova diretiva `transaction`. Este aspecto pode ser considerado positivo, pois as abordagens visam estender a capacidade de expressão de OpenMP, e não introduzir o uso de uma biblioteca ortogonal a esta linguagem. No entanto, o padrão OpenMP define a semântica de acesso a dados compartilhados por cláusulas associadas às diretivas de paralelização, como as cláusulas, `shared`, `private` e `reduce`. Assim, entende-se que uma semântica de acesso a dados manipulados por transações também deva ser definida da mesma forma, ou seja, sendo definida por cláusulas. Nesta tese, **`transaction`** é uma cláusula, não uma diretiva, uma vez que permite definir uma semântica ao compartilhamento de dados. Uma cláusula `transaction`, com semântica de acesso CRCW (*Concurrent Read, Concurrent Write*) para acesso aos dados, deve oferecer consistência no acesso a dados compartilhados tanto nas operações de leitura como nas operações de escrita.

4.2 Cláusula `transaction`

Este trabalho propõe uma nova cláusula para a especificação OpenMP a fim de permitir o uso de TM pelas seções paralelas de uma aplicação. Esta cláusula permite integrar TM ao modo como um código paralelo é implementado a partir do OpenMP. Desta forma, é possível o uso de TM sem alterar a forma de expressão do modelo de programação previsto pela API de OpenMP. A nova cláusula proposta é denominada `transaction`. A gramática (simplificada) desta extensão é apresentada na Figura 14.

```
1          #pragma omp parallel transaction [<cláusulas>]
2          { Bloco: Região Paralela }
3
4          #pragma omp <diretiva> transaction(<tdeclist>) [<cláusulas>]
5            { Bloco de Comandos }
6
7          Onde:
8
9              <diretiva> :: section | for | task
10             <tdeclist> :: <tdec> | <tdec> , <tdeclist>
11                 <tdec> :: <op> : <id> | <id>
12                   <op> :: [A|D ,] R | W | RW
13                   <id> :: identificador para a variável
14           <cláusulas> :: outras cláusulas OpenMP
```

Figura 14 – Interface proposta.

A cláusula `transaction` explicita o uso de Memória Transacional em um programa OpenMP com a extensão proposta. A cláusula `transaction` associada à diretiva `parallel` indica que, nesta região paralela, tarefas poderão realizar acessos

à Memória Transacional. Nas demais diretivas, a cláusula `transaction` é parametrizada com uma lista de identificadores `<id>`. Sobre os identificadores, ressalta-se que estes devem pertencer ao escopo das tarefas criadas. Caso algum identificador corresponda a um ponteiro, é considerado que a variável transacionada é aquela referenciada pelo ponteiro. Ou seja, caso o valor do ponteiro seja alterado, a alteração do ponteiro não é realizada de forma transacional, nem os acessos à nova posição referenciada.

O operador designado por `<op>` identifica o modo de acesso preferencial, conforme discutido na sequência, à variável transacional. Este acesso pode ser caracterizado como preferencial em leitura, escrita ou leitura e escrita, na utilização dos respectivos operadores: `R`, `W` ou `RW`. Na ausência da especificação do operador, é assumido que a transação opera o dado em leitura e escrita.

Outro modificador de acesso pode ser dado pelas opções `A` (*adopt*, opção default) e `D` (*defer*). Este modificador indica que se a tarefa que está sendo criada irá realizar, ela própria, a transação sobre o dado ou se uma tarefa aninhada é que realizará a transação. Este modificador é discutido adiante, neste capítulo.

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 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. Esta diretiva, em particular, é discutida na sequência deste capítulo, quando do abordada a questão da recursividade.

Visando um melhor detalhamento, a proposta foi exemplificada por meio da versão OpenMP dos problemas Cowichan. Todos os problemas Cowichan que permitiram a codificação com TM foram implementados utilizando a proposta aqui apresentada. Estes mesmos problemas também foram implementados aplicando-se as interfaces propostas nos trabalhos de (MILOVANOVIĆ et al., 2008) e (WONG et al., 2014), além de outras duas com OpenMP utilizando os suportes de Memória Transacional em TinySTM e GCC-TM.

Todas as versões, juntamente com a sequencial e OpenMP originais do Cowichan, estão disponíveis neste repositório Git[2]. Na figura 15 há uma comparação entre (a) o código OpenMP do benchmark *norm* implementado com OpenMP puro e (b) sua versão com a cláusula `transaction`.

[2]https://github.com/adjardim/OpenMP_TM_TinySTM

```
1
2  void findMinMax(PointVector pts, int n,
3                  Point* minPt, Point* maxPt) {
4     PointVector minPts=NULL, maxPts=NULL;
5     int n_th = omp_get_max_threads();
6     try {
7         minPts = NEW_VECTOR_SZ(Point, n_th);
8         maxPts = NEW_VECTOR_SZ(Point, n_th);
9     } catch (...) {out_of_memory();}
10
11    #pragma omp parallel
12    {
13        int th_num = omp_get_thread_num();
14        minPts[th_num].x = pts[0].x;
15        minPts[th_num].y = pts[0].y;
16        maxPts[th_num].x = pts[0].x;
17        maxPts[th_num].y = pts[0].y;
18        #pragma omp for schedule(static)
19        for(int i = 1; i < n; i++) {
20            if(pts[i].x < minPts[th_num].x)
21                minPts[th_num].x = pts[i].x;
22            if(pts[i].y < minPts[th_num].y)
23                minPts[th_num].y = pts[i].y;
24            if(pts[i].x > maxPts[th_num].x)
25                maxPts[th_num].x = pts[i].x;
26            if(pts[i].y > maxPts[th_num].y)
27                maxPts[th_num].y = pts[i].y;
28        }
29    }
30    minPt->x = minPts[0].x; minPt->y = minPts[0].y;
31    maxPt->x = maxPts[0].x; maxPt->y = maxPts[0].y;
32    for(int i = 0; i < n_th; i++) {
33        if (minPt->x > minPts[i].x)
34            minPt->x = minPts[i].x;
35        if (minPt->y > minPts[i].y)
36            minPt->y = minPts[i].y;
37        if (maxPt->x < maxPts[i].y)
38            maxPt->x = maxPts[i].x;
39        if (maxPt->y < maxPts[i].y)
40            maxPt->y = maxPts[i].y;
41    }
42    delete[] minPts; delete[] maxPts;
43}
```

(a)

```
1
2  void findMinMax(PointVector pts, int n,
3                  Point* minPt, Point* maxPt) {
4
5     minPt->x = pts[0].x;
6     minPt->y = pts[0].y;
7     maxPt->x = pts[0].x;
8     maxPt->y = pts[0].y;
9
10    //Laço paralelo com a cláusula transaction
11
12    #pragma omp parallel transaction
13    {
14    #pragma omp for schedule(static) \
15    transaction(RW : minPT, RW : maxPT)
16    for(int i = 1; i < n; i++) {
17        if(pts[i].x < minPt->x)
18            minPt->x = pts[i].x;
19        if(pts[i].y < minPt->y)
20            minPt->y = pts[i].y;
21        if(pts[i].x > maxPt->x)
22            maxPt->x = pts[i].x;
23        if (pts[i].y > maxPt->y)
24            maxPt->y = pts[i].y;
25     }
26    }
27}
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43//FIM
```

(b)

Figura 15 – Comparação do código OpenMP do benchmark *norm*: (a) OpenMP puro. (b) Com a cláusula `transaction`.

É possível observar nos códigos para as aplicações *norm* e *thresh* que as versões empregando a cláusula `transaction` produzem códigos menores em relação ao original. Isto decorre da eliminação do código que lida com os *threads* (linhas 4 a 13 e 30 a 42 da Figura 15(a)) que passa a ser realizado de forma implícita pela própria transação realizada no acesso às variáveis compartilhadas (`minPt` e `maxPt`). Também observa-se, no código para o benchmark *norm* (Figura 15(b)) ser possível utilizar diretamente a diretiva `#pragma omp parallel for`, deixando mais clara a existência de um laço paralelo.

A Figura 16 exibe um segundo exemplo de implementação a partir da cláusula `transaction`, desta vez com o benchmark *thresh*. Neste exemplo também observa-se que a cláusula `transaction` permite reduzir o código pelo o uso da TM para controlar os acessos a um vetor (identificador `hist`). Na codificação OpenMP sem transação, é necessário fazer uso de uma matriz para proceder os cálculos localmente em cada *thread* (linhas 11 a 24 da Figura 16(a)).

A cláusula `transaction` visa, além de simplificar o código paralelo, evitar o uso de seções críticas ou bloqueios, como os suportados pela diretiva `critical`. Outro recurso proposto é relacionado a possibilidade de especificar o modo de acesso (`<op>`) preferencial aos identificadores que serão transacionados nas tarefas que serão criadas. Nos exemplos de código exibidos nas figuras 15(b) e 16(b), os acessos foram definidos como sendo, no primeiro caso `<RW>`, indicando que as variáveis serão acessadas 50% por leituras e 50% por escritas e, no segundo, como sendo `<W>`, ou seja, majoritariamente de escrita. Caso o modo de acesso fosse definido como `<R>`, a informação indicaria que o identificador seria utilizado, na tarefa, majoritariamente em leitura. A definição do modo de acesso prioritário permite, a critério do programador, otimizar a escolha das técnicas de versionamento e detecção de conflito pela TM, instruindo o ambiente de execução sobre como os identificadores são utilizados.

4.2.1 Modelo de Memória: OpenMP com Transações

A introdução de um mecanismo de Memória Transacional em OpenMP implica que um novo estrato de memória seja inserido no mapa de memória de um programa em execução. O novo desenho de memória encontra-se na Figura 17. Onde programa apresentado na Figura 17.(a) manipula a variável identificada por `b` em Memória Transacional. O novo estrato é destacado na Figura 17.(b), representa a Memória Transacional, no qual as variáveis transacionadas são mantidas.

Deve ser observado que a Memória Transacional sobre o identificador `b` opera na execução das tarefas criadas pela diretiva `task`. A consistência do dado, na função `main`, somente é garantido após o retorno da diretiva `taskwait`. Situação semelhante ocorre com o uso da cláusula `transaction` associado à cláusula `nowait` com

```
1
2  void Cowichan::thresh(IntMatrix mat,
3             BoolMatrix mask) {
4     int* hist = NULL; int i, j, r, c; int vMax;
5     int n_th = omp_get_max_threads();
6
7  //código ocultado com inicialização vMax..
8     try {
9         hist = NEW_VECTOR(int, vMax+1);
10    } catch (...) {out_of_memory();}
11    int** histLc = NULL;
12    try {
13        hLoc = NEW_VECTOR(int*, n_th);
14        for(i = 0; i < n_th; i++)
15            hLoc[i] = NEW_VECTOR(int, vMax+1);
16    } catch (...) {out_of_memory();}
17
18    #pragma omp parallel for \
19    schedule(static) private(j)
20    for(i = 0; i <= vMax; i++) {
21        hist[i] = 0;
22        for (j = 0; j < n_th; j++)
23            hLoc[j][i] = 0;
24    }
25    #pragma omp parallel
26    {
27        int th_num = omp_get_thread_num();
28        #pragma omp for schedule(static)
29        for(r = 0; r < nr; r++) {
30            #pragma omp for schedule(static)
31            for (c = 0; c < nc; c++)
32              hLoc[th_num][MATRIX_RECT(mat, r, c)]++;
33        }
34    }
35    for(i = 0; i < n_th; i++) {
36        for (j = 0; j <= vMax; j++)
37            hist[j] += hLoc[i][j];
38        delete [] hLoc[i];
39    }
40    delete [] hLoc;
41 //código ocultado continuação função..
42 }
```

(a)

```
1
2  void Cowichan::thresh(IntMatrix mat,
3         BoolMatrix mask) {
4     int* hist = NULL; int i, r, c; int vMax;
5
6  //código ocultado com inicialização vMax..
7     try {
8         hist = NEW_VECTOR(int, vMax + 1);
9     } catch (...) {out_of_memory();}
10
11    #pragma omp parallel for \
12    schedule(static)
13    for (i = 0; i <= vMax; i++)
14        hist[i] = 0;
15
16    //Laço paralelo com a cláusula transaction
17
18    #pragma omp parallel for \
19    schedule(static)
20    for (r = 0; r < nr; r++) {
21        #pragma omp for schedule(static)\
22        transaction(W : hist)
23        for (c = 0; c < nc; c++)
24            hist[MATRIX_RECT(mat, r, c)]++;
25    }
26 //código ocultado continuação função..
27
28 }
29
30
31
32
33
34
35
36
37
38
39
40
41
42 //FIM
```

(b)

Figura 16 – Comparação do código OpenMP do benchmark *tresh*: (a) *tresh* Cowichan (b) *tresh* `transaction`.

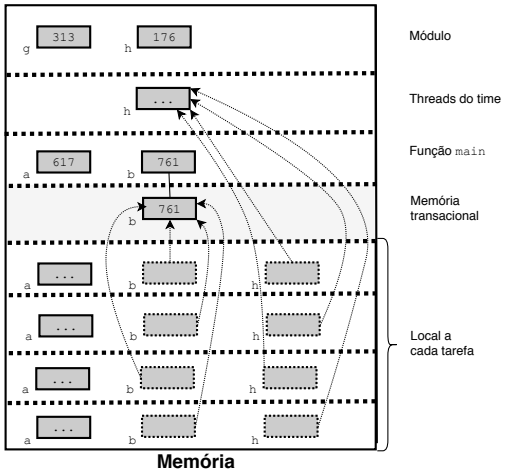
```
#include <omp.h>

int g = 313, h = 176;

int main( int argc, char **argv ) {
 int a = 617, b = 761;

 #pragma omp threadprivate(h)
 #pragma omp parallel num_threads(4)
 {
  #pragma omp task private(a), transaction(b)
   {
    ...
    ...
    ...
   }
  #pragma omp taskwait
 }
 return 0;
}
```

(a)



(b)

Figura 17 – Exemplo de um programa OpenMP com Memória Transacional (a) e o mapa de memória obtido em tempo de execução (b).

qualquer outra diretiva de paralelização.

4.3 Análise do código obtido

Como forma de comparar a presente proposta da cláusula `transaction` com diferentes abordagens que também propuseram o uso de TM em OpenMP, utilizou-se métricas de análise de código. Estas métricas foram aplicadas sobre implementações de programas pertencentes ao benchmark Cowichan, desenvolvido para avaliar o poder de expressão de interfaces para programação concorrente e paralela. Neste sentido, as comparações tiveram por base os códigos OpenMP originais dos problemas Cowichan, as versões empregando as extensões OpenMP propostas por (MILOVANOVIĆ et al., 2008) e (WONG et al., 2014), e as versões com OpenMP utilizando os suportes de Memória Transacional em TinySTM e GCC-TM. Apenas seis problemas do benchmark Cowichan, de um total de 14, foram considerados nesta avaliação, uma vez que nos demais casos TM não se mostrou aplicável. Os seis problemas considerados foram: hull, norm, outer, sor, thresh e vecdiff.

As métricas aplicadas para mensurar as diferenças no código foram concebidas conforme a abordagem GQM (*Goal Question Metric*). Tal abordagem é baseada na suposição de que para uma organização mensurar de forma proposital, ela deve primeiro especificar os objetivos para si mesma e seus projetos, em seguida, deve rastrear esses objetivos aos dados que se destinam a definir estes objetivos operacionalmente. Além disso, fornecer uma estrutura para interpretar os dados em relação aos objetivos declarados (CALDIERA; ROMBACH, 1994). Seguindo a abordagem GQM, foram elencadas quatro questões. O objetivo (goal) é *caracterizar o uso das diferentes interfaces para TM em OpenMP*. As métricas estão apresentadas na

Tabela 2 – Questões e Métricas aplicadas na comparação.

Q1. Qual é o tamanho do código?	
Métrica	NLC - Número de linhas do código
Definição	Número total de linhas de código com instruções e chamadas da API.
Comentário	Este número não inclui linhas em branco e de comentário no arquivo fonte.
Q2. Quantos recursos diferentes da API de OpenMP são utilizadas?	
Métrica	QRU - Quantidade de recursos utilizados.
Definição	Informa quantos recursos distintos da API de OpenMP são necessários para realizar a implementação.
Comentário	São inclusas na contagem todas as diretivas, cláusulas e chamadas de funções definidas para OpenMP.
Q3. Quantas diretivas de paralelização são invocadas no código?	
Métrica	QIDP - Quantidade de invocações a diretivas de paralelização.
Definição	Número de ocorrências totais de diretivas OpenMP para criação de tarefas.
Comentário	Caracteriza a quantidade de blocos paralelos necessários na implementação.
Q4. Qual é a proporção da quantidade de recursos OpenMP utilizados no tamanho do código?	
Métrica	Rel1 - Relação entre Tamanho do Código e Recursos Utilizados.
Definição	Rel1 = QRU / NLC
Comentário	Maior o valor apresentado, menor, proporcionalmente, é a quantidade de recursos utilizados.
Q5. Qual é a proporção entre zonas paralelas e o tamanho do código?	
Métrica	Rel2 - Relação Tamanho do Código e Número de Diretivas de Paralelização.
Definição	Rel2 = QIDP / NLC
Comentário	Caracteriza a quantidade de blocos paralelos necessários na implementação.

Tabela 2.

Na Tabela 3 são apresentados os dados resultantes da análise dos códigos. Eles sugerem que a cláusula `transaction` aqui proposta proporciona uma redução no número de linhas (NLC), diretivas (QRU) e invocações de diretivas (QIDP) em comparações com o código OpenMP original e com as versões empregando as extensões OpenMP propostas por (MILOVANOVIĆ et al., 2008) e (WONG et al., 2014).

Redução no tamanho do código também pode é observada em todos os problemas Cowichan utilizando a extensão proposta por (MILOVANOVIĆ et al., 2008) e em alguns problemas aplicando a proposta de (WONG et al., 2014). Em relação à quantidade de recursos utilizados, a interface apresentada requer, nos problemas considerados, a mesma quantidade de recursos diferentes que em (WONG et al., 2014) e as três propostas de extensões para manipulação de Memória Transacional oferecem um código mais enxuto que OpenMP padrão. Os dados utilizando OpenMP com as ferramentas de suporte à Memória Transacional também mostraram redução geral nos valores.

Como indicativo geral, há a percepção de que a abstração fornecida por mecanismos de manipulação permite reduzir tanto o tamanho do código quanto a quantidade de recursos de programação da ferramenta de programação utilizada.

A presente proposta diferencia-se das anteriores pela forma como são definidos os identificadores que devem ser transacionados. Na proposta de (WONG et al., 2014), não é possível fazer essa definição. Assim, ou utiliza-se a transação em toda a seção paralela ou adiciona-se uma diretiva exclusiva para a determinar a parte do código que deve contemplar TM. Já na proposta do Nebelung (MILOVANOVIĆ et al., 2008), pode-se informar as variáveis que serão transacionadas, porém isso deve ser feito utilizando outra cláusula. Além disso, nenhuma das outras abordagens possui

Tabela 3 – Comparação entre códigos OpenMP dos problemas Cowichan

Problema	hull	norm	outer	sor	tresh	vecdiff	hull	norm	outer	sor	tresh	vecdiff
Métrica	**OpenMP com cláusula transaction**						Código original OpenMP					
NLC	152	60	31	41	59	20	208	95	53	61	102	42
QRU	3	3	4	4	3	3	2	2	3	4	3	3
QIDP	3	2	3	1	7	1	5	3	4	2	9	2
QRU / NLC	0,02	0,05	0,13	0,1	0,05	0,13	0,01	0,02	0,06	0,7	0,03	0,07
QIDP / NLC	0,02	0,03	0,1	0,02	0,12	0,05	0,02	0,03	0,08	0,03	0,09	0,05
	OpenMP com (WONG et al., 2014)						OpenMP com Nebelung (MILOVANOVIĆ et al., 2008)					
NLC	155	60	34	44	59	23	152	60	31	41	59	20
QRU	3	3	4	4	3	3	4	4	5	4	4	4
NIDP	4	2	4	2	7	2	3	2	3	1	7	1
QRU / NLC	0,02	0,05	0,12	0,09	0,05	0,13	0,03	0,07	0,16	0,1	0,07	0,2
NIDP / NLC	0,03	0,03	0.12	0,05	0,12	0,09	0,02	0,03	0,1	0,02	0,12	0,05
	GCC __atomic						TinySTM					
NLC	167	68	38	48	70	28	180	78	45	55	86	35
QRU	3	2	3	3	2	3	3	2	3	3	2	3
QIDP	4	3	4	2	9	2	5	4	4	2	9	2
QRU / NLC	0,02	0,03	0,08	0,06	0,03	0,11	0,02	0,03	0,07	0,05	0,02	0,09
QIDP / NLC	0,02	0,04	0,11	0.04	0,13	0,07	0,03	0,05	0,09	0,04	0,1	0,03

recurso para especificar o modo de acesso (leitura ou escrita) para os identificadores da transação.

Em relação às implementações com TinySTM e GCC-TM, foi detectado um aspecto relevante que expõe a diferença de nível de abstração por estas com o provido pela interface proposta. As interfaces de programação providas por TinySTM e GCC-TM requerem que o programador informe o endereço de memória que será manipulado de forma transacional. Isso faz com que o uso de uma determinada variável, ou seja, de uma célula de memória, na Memória Transacional não seja dependente do escopo do identificador que o representa. Na interface proposta, por outro lado, o uso da variável em uma transação somente é permitido enquanto existir vínculo da variável com o identificador associado ao seu uso. No caso prático, havendo uma chamada a uma função em uma transação e esta função, de posse de um *alias*, seja pela manipulação de um ponteiro, de uma referência ou mesmo se tratando de uma variável de escopo global, para o dado transacionado efetua uma alteração neste dado, o efeito colateral não é percebido pela transação.

4.4 Impacto na Recursão

Os exemplos discutidos anteriormente não tratam algoritmos recursivos, suportados, por OpenMP, pela diretiva `task`. A opção pela implementação deste mecanismo recai na adoção do aninhamento fechado (*close nesting*). Esta política implica que, havendo uma transação interna a outra, esta pode ser abortada e, então, reiniciada, independente da transação envolvente. A publicação de suas alterações na memória, por consequência, são visíveis no momento em que a operação de *commit* é realizada, independente do sucesso ou falha da transação externa. Esta estratégia é interessante em um programa OpenMP considerando diferentes aspectos:

- Uma dependência explícita de dados entre a tarefa executando a transação envolvente e aquela executando a transação aninhada pode ser resolvida utilizando a diretiva `taskwait`;
- Em um momento anterior à invocação da diretiva `taskwait`, ambas tarefas, envolvente e aninhada, executam de forma concorrente, podendo resultar que a operação de *commit* realizada por uma delas afete os dados manipulados pela outra.
- Programas OpenMP recursivos podem gerar um grande número de tarefas de pequena granularidade e o custo de uma recuperação em cascata inviabilizaria o uso da técnica.

A interface proposta propõe o uso de um modificador de acesso para apoio a gestão do aninhamento das transações: `A` (*adopt*), indicando que a tarefa irá, ela própria manipular o dado de forma transacionada, e `D` (*defer*), indicando que a tarefa não realizará manipulação transacionada sobre o dado, mas que este dado será encaminhado, recursivamente, a outra tarefa, que terá à disposição os mesmos modificadores de acesso. A opção padrão é *adopt*.

A Figura 18 apresenta dois exemplos de criação de tarefas de forma recursiva e manipulação de dados de forma transacionada. Em ambos exemplos, a tarefa mais externa decrementa o dado compartilhado e as mais internas, incrementam. Na Figura 18.(a) existe a possibilidade de acesso concorrente ao dado por uma, ou ambas, tarefa interna e a externa. Como a tarefa externa indica que receberá o dado para ser manipulado sob o regime transacional (*adopt*), a manipulação é realizada de forma coordenada. Na Figura 18.(b), a tarefa externa deixa explícito que não irá manipular o dado de forma transacional, mas que uma de suas descendentes poderá fazê-lo. Neste segundo caso, a tarefa externa tem acesso consistente ao dado por realizar o acesso após a sincronização do término das tarefas internas.

Nos trechos de código dos exemplos na Figura 18, o resultado final na variável `a` será o mesmo. A opção em utilizar o modo *defer* é uma alternativa que o programador tem para reduzir o impacto das operações em Memória Transacional na execução de seu programa.

4.5 Prototipação

A prototipação da extensão proposta explora a interface *Vanilla*-TM. Esta interface consiste em uma linguagem intermediária concebida para permitir que a extensão proposta possa ser suportada por diferentes ferramentas (bibliotecas) de suporte a Memória Transacional em Software. O modelo arquitetural é apresentado na Figura

```
int a;
...
#pragma omp task transaction(A:a)
{
 #pragma omp task transaction(A:a)
 {
  a++;
 }
 #pragma omp task transaction(A:a)
 {
  a++;
 }
 a--;
#pragma omp taskwait
}
#pragma omp taskwait
```

(a)

```
int a;
...
#pragma omp task transaction(D:a)
{
 #pragma omp task transaction(A:a)
 {
  a++;
 }
 #pragma omp task transaction(A:a)
 {
  a++;
 }
#pragma omp taskwait
 a--;
}
#pragma omp taskwait
```

(b)

Figura 18 – Transações em tarefas aninhadas

19. O programa de aplicação é desenvolvido em OpenMP, utilizando todos os recursos disponíveis na especificação OpenMP e contando com a nova cláusula `transaction`. Após a tradução para a linguagem intermediária, o suporte operacional à cláusula `transaction` é realizado pela ferramenta de STM selecionada, sem interferir no modo de operação de OpenMP e sem utilizar nenhuma facilidade oferecida por este padrão.

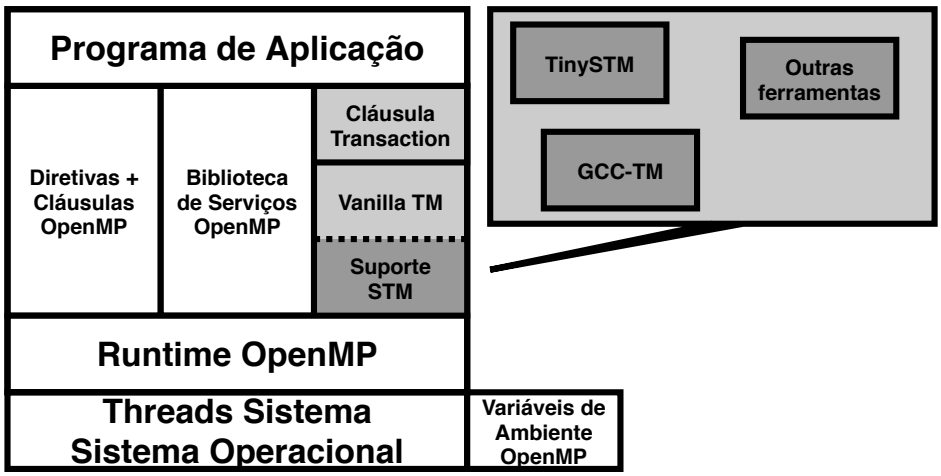


Figura 19 – Arquitetura do ambiente proposto sobre OpenMP.

Na Figura 20 é apresentada a inserção da etapa, no processo de geração do executável, responsável pelo tratamento da cláusula `transaction`. Um procedimento de tradução de código identifica, no programa fonte da aplicação, o uso da cláusula proposta, manipulando o código de forma a obter um novo programa OpenMP com chamadas aos serviços da ferramenta adotada para suporte à TM.

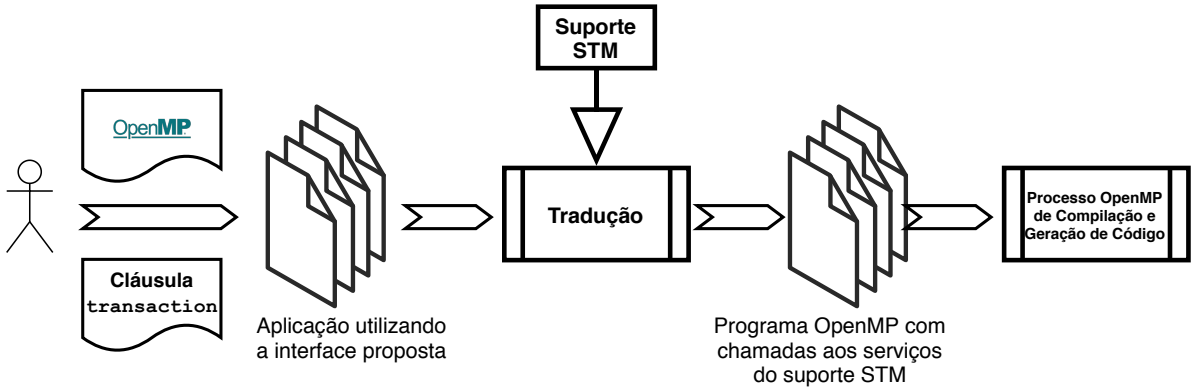


Figura 20 – Processo para obtenção do código.

4.5.1 Interface *Vanilla*-TM

A interface *Vanilla*-TM é implementada na forma de um conjunto de macros com o seguinte conjunto de serviços:

- `vtm_t* vtm_init( vtm_attr_t* vtm );`
  Inicializa o ambiente de Memória Transacional, informando políticas para manipulação da Memória Transacional, conforme possibilidades oferecidas pela ferramenta utilizada no suporte a STM. Retorna um descritor para o manipulador de Memória Transacional.

- `void vtm_finalize();`
  Finaliza o ambiente de Memória Transacional. Não há comportamento especificado na realização desta operação caso o ambiente não esteja ativo.

- `void vtm_thread_init( vtm_t* tm );`
  Cada thread deve invocar esta primitiva uma única vez, de forma a associá-lo ao mecanismo de gestão da Memória Transacional. Caso `tm` seja `NULL`, esta primitiva não realiza nenhuma operação.

- `void vtm_thread_finish( vtm_t* tm );`
  Ao ser invocada por um thread, o thread deixa de ser considerado no mecanismo de Memória Transacional. Caso `tm` seja `NULL`, esta primitiva não realiza nenhuma operação.

- `vtm_tx_t* vtm_start( vtm_dataset_t* dataSet );`
  Determina o início de uma transação e identifica o conjunto de dados monitorados. O parâmetro de entrada da função contém a lista de endereços de memória de variáveis monitoradas na transação e seus respectivos modos de acesso. Retorna um descritor para a transação criada.

- `void vtm_commit( vtm_tx_t* tm );`
  Caso a transação conclua com sucesso, prossegue a execução. Caso contrário, realiza nova entrada. Esta primitiva não tem retorno, pois apenas retorna quando a operação for bem sucedida. Não há comportamento especificado na realização desta operação em uma transação não aberta.

- `vtm_uint_t vtm_read( vtm_tx_t* tm, vtm_uint_t* addr);`
  Realiza acesso em leitura do conteúdo de memória referenciado pelo endereço informado em `addr`. Caso `addr` não seja manipulado na transação, o retorno ainda é representado pelo conteúdo da memória referenciada, no entanto, o acesso não se dá de forma transacional.

- `void vtm_write( vtm_tx_t* tm, vtm_uint_t value,`
  `vtm_uint_t* addr);`
  Realiza acesso na posição de memória indicada por `addr` para escrita de `value`. Caso `addr` não seja manipulado na transação, a escrita do dado ainda ocorre, embora não de forma transacional.

A opção por manipular o *dataset* com o tipo `void*` oferece a possibilidade do uso de *Vanilla*-TM em programas C. De forma objetiva, `vtm_tm_t` consiste no descritor de uma transação e `vtm_dataset_t` no descritor do conjunto de dados manipulados na transação, implementados como tipos de dados opacos. As primitivas para manipulação de `vtm_dataset_t`, pelo programa de aplicação, são as seguintes:

- `int vtm_dataset_init( vtm_dataset_t* dtas );`
  Inicializa um descritor de dados de transição. Retorna 0 (zero) caso a operação tenha sido bem sucedida ou um valor diferente de 0 em caso de falha na inicialização.

- `int vtm_dataset_pack( vmt_dataset_t* dtas, char am,`
  `vtm_uint_t* dta );`
  Adiciona ao descritor de dados uma nova variável, no endereço `dta` definida no modo de acesso `am`. Retorna 0 (zero) caso a operação tenha sido bem sucedida ou um valor diferente de 0 em caso de falha na inclusão.

- `int vtm_dataset_packarray( vmt_dataset_t* dtas, int size,`
  `char* am, vtm_uint_t** dta );`
  Adiciona ao descritor de dados `size` variáveis, endereçáveis pelo vetor `dta` com os respectivos modos de acesso identificados no vetor `am`. Retorna 0 (zero) caso a operação tenha sido bem sucedida ou um valor diferente de 0 em caso de falha na inclusão.

- `void vtm_dataset_destroy( vtm_dataset_t* dtas );`
  Destroi um descritor de transação, permitindo seu reuso. Não é definido comportamento para falha nesta operação.

Importante observar que a biblioteca *Vanilla-TM* não é efetivamente uma nova biblioteca para manipulação de TM em programas C. Trata-se de uma representação intermediária para permitir a portabilidade das implementações realizadas entre diferentes ferramentas de programação com suporte à TM. Dentre as ferramentas com suporte ao modelo de TM em Software, esta tese 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.

4.5.2 Tipos de dados e regras de tradução

A interface *Vanilla*-TM consiste em uma linguagem intermediária. Sua especificação visa atender as necessidades de diferentes ferramentas oferecendo recursos para exploração de Memória Transacional em Software. Os tipos de dados manipulados nesta linguagem intermediária visam permitir a comunicação dos dados manipulados efetivamente por estas bibliotecas. Estes tipos de dados são manipulados de forma opaca, ou seja, seus campos não devem ser manipulados por primitivas, nunca sendo acessados seus campos diretamente.

`vtm_uint_t`: Tipo inteiro com sinal, capaz de armazenar um valor inteiro ou um endereço de memória. Corresponde a uma palavra da arquitetura e consiste na unidade básica de manipulação de dados na Memória Transacional.

`vtm_attr_t`: Estrutura de dados contendo parâmetros de configuração a serem repassados para inicialização da ferramenta de suporte à memória transacional em software.[3]

`vtm_t`: Estrutura de dados contendo configurações de inicialização e informações sobre o mecanismo de gestão de Memória Transacional instanciado. Uma única estrutura deste tipo é instanciada, no início do programa.

`vtm_tm_t`: Estrutura de dados contendo informações sobre o suporte à Memória Transacional sobre cada thread em execução. São instanciadas tantas instâncias desta estrutura de dados quanto forem o número de threads em execução em um programa.

[3] Esta estrutura de dados não foi implementada no protótipo, sendo as avaliações realizadas com a configuração padrão das ferramentas utilizadas. Seu efetivo uso requer implantação de primitivas adequadas para manipulá-la.

vtm_tx_t: Estrutura de dados contendo informações sobre uma determinada transação.

vtm_dataset_t: Estrutura de dados contendo informações específicas sobre a transação em execução em uma tarefa. São instanciadas tantas instâncias desta estrutura quanto forem o número de tarefas manipulando dados na forma transacional.

Ao apresentar as estruturas de dados, é importante mostrar um aspecto relevante em relação aos tipos dos dados transacionáveis. Em função das características das ferramentas trabalhadas até o momento, GCC-TM e TinySTM, os experimentos realizados, e as implementações consideradas, manipulam dados acessíveis atomicamente. Isso corresponde a dados cujo comprimento seja o de uma *palavra* da arquitetura (32 ou 64 bits). Este tipo de dado é representado pelo tipo `vtm_uint_t`.

A Figura 21 ilustra o processo de tradução do código. É apresentado, na Figura 21.(a) o código escrito com a interface proposta e, na Figura 21.(b) seu código traduzido. A tradução realizada é detalhada na sequência.

Durante o processo de tradução do código, chamadas às primitivas `vtm_init` e `vtm_finalize` são inseridas como, respectivamente, a primeira e a última operação do programa. Sua função é inicializar e finalizar as operações da ferramenta de suporte ao ambiente de Memória Transacional. O protótipo construído não utiliza a parametrização da inicialização das bibliotecas, utilizando as configurações padrão por elas propostos. O parâmetro de entrada da primitiva `vtm_init` não é, portanto, utilizado. Não há correspondente a esta tradução na interface proposta. O código gerado é apresentado na Figura 21.(b), linhas 3, 10 e 60.

A inicialização do ambiente de suporte à Memória Transacional em cada thread é realizado pela primitiva `vtm_thread_init`. A finalização, em cada thread, pela primitiva `vtm_thread_finish`. A invocação a estes códigos é inserida, no processo tradução, ao ser identificada a cláusula `transaction` associada ao pragma `parallel`. Na Figura 21.(a), o código de interesse encontra-se corresponde na linha 6. O código traduzido, Figura 21.(b), introduz as linhas 7, 10 e 58.

No código da Figura 21.(a) há instanciação de três tarefas, nas linhas 8, 10 e 14. Exemplificando a tradução com o código correspondente à tarefa da linha 14. O código traduzido encontra-se na Figura 21.(b), linhas 28-42. Para manter o encapsulamento, um novo bloco de comandos é definido (limites nas linhas 30 e 41). Neste bloco, é instanciada as estruturas de dados manipuladas pelo suporte de Memória Transacional (linhas 31 e 32) e uma variável auxiliar (linha 33) para realizar as operações sobre o dado transacionado (linha 38). A transação está definida entre as linhas 36 e 40. O acesso efetivo à Memória Transacional se dá nas linhas 37 e 39.

```
1
2  #include <vanilla.h>
3
4  int main() {
5  int a;
6  #pragma omp parallel transaction
7  {
8   #pragma omp task transaction(A:a)
9   {
10   #pragma omp task transaction(A:a)
11   {
12    a++;
13   }
14   #pragma omp task transaction(A:a)
15   {
16    a++;
17   }
18   a--;
19   #pragma omp taskwait
20  } // task externa
21  #pragma omp taskwait
22 } // parallel
23 return 0;
24}
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62//FIM
```

(a)

```
1
2  #include <vanilla.h>
3  vtm_t tm;
4
5  int main() {
6  int a;
7  tm = vtm_init( NULL );
8  #pragma omp parallel
9  {
10  vtm_thread_init(&tm);
11  #pragma omp task
12  {
13   #pragma omp task
14   {
15   {
16    vtm_data_set_t dta002;
17    vtm_tx_t tx002;
18    vtm_uint_t tmp002;
19    vtm_dataset_init( &dta002 );
20    vtm_dataset_pack( &dta002, RW, &a );
21    tx002 = vtm_start( &dta002 );
22    tmp002 = vtm_read( &tx002, &a );
23    tmp002++;
24    vtm_write( &tx002, tmp002, &a );
25    vtm_commit( &tx002 );
26   }
27   }
28   #pragma omp task
29   {
30   {
31    vtm_data_set_t dta003;
32    vtm_tx_t tx003;
33    vtm_uint_t tmp003;
34    vtm_dataset_init( &dta003 );
35    vtm_dataset_pack( &dta003, RW, &a );
36    tx003 = vtm_start( &dta003 );
37    tmp003 = vtm_read( &tx003, &a );
38    tmp003++;
39    vtm_write( &tx003, tmp003, &a );
40    vtm_commit( &tx003 );
41   }
42   }
43   {
44    vtm_data_set_t dta001;
45    vtm_tx_t tx001;
46    vtm_uint_t tmp001;
47    vtm_dataset_init( &dta001 );
48    vtm_dataset_pack( &dta001, RW, &a );
49    tx001 = vtm_start( &dta001 );
50    tmp001 = vtm_read( &tx001, &a );
51    tmp001--;
52    vtm_write( &tx001, tmp001, &a );
53    vtm_commit( &tx001 );
54   #pragma omp taskwait
55  }
56  } // task externa
57  #pragma omp taskwait
58  vtm_thread_finish(&tm);
59 } // parallel
60 vtm_finish();
61 return 0;
62}
```

(b)

Figura 21 – Resultado da tradução de código (a) Código de entrada. (b) Tradução para *Vanilla*-TM.

O código correspondente ao bloco iniciando na linha 43 na Figura 21.(b) corresponde à tarefa externa do bloco aninhado. No protótipo construído assume-se que uma tarefa que adquira uma variável para realizar operação transacional e crie outras tarefas que também acessam o dado de forma transacional, somente irá realizar sua própria transação após a criação das suas tarefas descentes. Conforme ilustrado na Figura 21.(a).

4.5.3 Tradutor para *Vanilla*-TM

O processo de tradução dos códigos apresentados neste texto, incluindo aqueles utilizados nas avaliações de desempenho descritas no Capítulo 5, foi realizada de forma manual. Em uma etapa de trabalho posterior, iniciou-se o desenvolvimento, na linguagem Haskell, de um programa tradutor, viabilizando, no momento, a tradução parcial de programas OpenMP utilizando a interface proposta. Esta seção apresenta este tradutor.

O tradutor implementa um subconjunto da especificação da linguagem C, tratando a função `main`, demais funções, estruturas de controle e de iteração, expressões lógicas e aritméticas e alguns tipos básicos. De OpenMP, o tradutor considera as cláusulas e a extensão para transações propostas. O tradutor opera sobre uma árvore sintática abstrata do programa. Quando identificadas as construções sintáticas propostas utilizando a cláusula `transaction`, ações de tradução são processadas.

1. Na identificação da construção `parallel transaction`, são tomadas as seguintes ações para geração do código em *Vanilla*-TM:

   (a) A cláusula `transaction` é suprimida da diretiva `parallel`;

   (b) É introduzido código com chamadas dos serviços *Vanilla*-TM para inicialização desta biblioteca no ponto que antecede o ingresso na região paralela indicado pela diretiva `parallel` e para finalização imediatamente ao final deste bloco.

2. Na identificação da cláusula `transaction` associada às demais diretivas:

   (a) Remove a cláusula `transaction` da diretiva, armazenando as informações sobre os identificadores manipulados e os modos de acesso a eles associados; a cláusula `shared` é associada à diretiva, contendo a lista de identificadores originalmente presentes na cláusula `transaction`;

   (b) Introduz as chamadas necessárias à *Vanilla*-TM no início e no final do bloco para vincular os endereços de memória dos identificadores das variáveis transacionadadas com o mecanismo de controle da transação;

(c) Percorre o bloco para introduzir instruções de acesso à Memória Transacional quando necessário:

i. Gera as variáveis temporárias para a lista de identificadores obtidos na cláusula `transaction`;

ii. Em comandos de atribuição na forma $l - value := r - value$, onde $l - value$ é uma variável transacionada, $l - value$ é substituído por uma variável temporária e, no final do bloco, é gerada a escrita do conteúdo da variável temporária na Memória Transacional e atualizada a variável original em $l - value$;

iii. Em comandos de atribuição na forma $l - value := r - value$ verifica, também, quais identificadores em $r - value$ que representam dados transacionados, introduzindo código de acesso, em leitura, dos respectivos dados nos temporários que os representam, indroduzindo código *Vanilla*-TM para realizar as operações de leitura destes dados. Então, substitui em $r - value$ todas as ocorrências dos identificadores a variáveis transacionadas pelos seus respectivos temporários.

iv. Nas demais construções que suportam expressões, como `if`, `for` e `while`, realiza o mesmo procedimento, introduzindo operações de leitura às variáveis sobre temporários, utilizando os temporários para realizar o cálculo da expressão.

4.5.4 Considerações sobre a prototipação

A abordagem de prototipação adotada para validação deste trabalho introduziu uma camada de abstração para acesso à Memória Transacional sobre o suporte de execução oferecido por OpenMP. Esta camada de abstração é representada pela biblioteca *Vanilla*-TM, a qual permite ser associada a diferentes bibliotecas de gestão de Memória Transacional, não sendo, portanto, dependente de um único suporte à Memória Transacional. Esta abordagem para a prototipação mostrou-se suficiente para permitir tanto a validação do poder de expressão da interface proposta como a obtenção de códigos executáveis, passíveis de análise de desempenho. Estes resultados são apresentados no Capítulo 5.

Outra abordagem possível para a prototipação é pela inclusão dos recursos de forma nativa no próprio suporte de OpenMP. Esta alternativa foi avaliada e considerada não prioritária na presente etapa de desenvolvimento da tese, na qual é proposta uma extensão a OpenMP e avaliado seu poder de expressão. Em conjunto, para tomada desta decisão, foram considerados dois aspectos práticos. O primeiro corresponde ao fato de que uma solução ao suporte à abstração de Memória Transacional deveria ser contemplada no runtime de OpenMP. A introdução de um suporte

real para Memória Transacional neste *runtime* já consistiria em um esforço equivalente a uma nova tese. Por outro lado, uma implementação *naive*, utilizando seções críticas para controle de acesso, não ofereceria vantagens reais em relação à metodologia adotada.

A biblioteca *Vanilla*-TM consiste em uma linguagem intermediária, explicitando as operações em Memória Transacional a serem utilizadas em um programa escrito em OpenMP utilizando a interface proposta. Esta biblioteca não é disponibilizada diretamente ao programador, mas suas chamadas introduzidas nos pontos específicos onde as operações sobre Memória Transacional são demandadas em um programa fonte por um processo de tradução de código. Por exemplo, o código apresentado na Figura 21.(b) é obtido a partir da tradução do código fonte apresentado na Figura 21.(a).

A interface da biblioteca *Vanilla*-TM foi concebida para contemplar uma ampla variedade de recursos na programação que possam ser oferecidos por ferramentas de suporte à Memória Transacional. Alguns destes recursos, no entanto, não encontram-se disponíveis nas ferramentas que foram efetivamente utilizadas, como aqueles que permitem indicar o modo de acesso aos dados.

4.6 Considerações

A proposta de interface para Memória Transacional apresentada estende a capacidade de expressão da linguagem base, no caso, OpenMP, pela incorporação de um novo recurso na sua interface de programação e não pelo uso de uma biblioteca de serviços. O ganho obtido neste ponto é associar a semântica das operações sobre Memória Transacional com a oferecida pelas outras operações já disponíveis em OpenMP, além de garantir a compatibilidade dos dados manipulados no programa com os serviços sobre a Memória Transacional. Faz-se notar que os trabalhos identificados como relacionados ((MILOVANOVIĆ et al., 2008) e (WONG et al., 2014)) não abordam o uso de Memória Transacional associado à diretiva `task`, por consequência, não abordam explicitamente o uso desta diretiva ou tecem considerações sobre a implicação da recursão nas soluções propostas.

Outro aspecto da presente proposta é permitir introduzir informações sobre a manipulação das variáveis nas transações, informando se seu uso se dará em operações de leitura, escrita ou leitura/escrita. Esta informação permite que, caso disponível, o *runtime* de suporte à Memória Transacional opte pela política mais adequada ao caso. Este recurso contempla estratégia similar à apresentada em (MEDIĆ et al., 2020), no entanto, nas ferramentas utilizadas para prototipação, TinySTM e GCC-TM, não é possível diferenciação nas políticas aplicadas a cada transação. Sua funcionalidade não foi, portanto, avaliada.

A implementação desta extensão nas aplicações do Cowichan permitiu compará-la a outros estudos que também propuseram integrar TM ao OpenMP, seja por extensões à esta linguagem ou pelo uso de bibliotecas de serviços. Apesar das semelhanças entre as implementações no código base das aplicações, a cláusula OpenMP aqui proposta está mais integrada à API, pois aplica a mesma sintaxe para definição dos identificadores já utilizada por outras cláusulas já existentes. Além disso, ela também traz outras vantagens em relação às outras abordagens, como a possibilidade de definição do modo de acesso das variáveis na transação. Dentre os resultados obtidos, cita-se a redução do número de linhas de código ao empregar a cláusula `transaction`, graças a expressividade do recurso proposto.

Como consideração final, aponta-se que no modelo de TM, o ambiente de execução monitora todos os endereços acessados em um bloco transacional e o modo de acesso (R/W). As ferramentas que implementam TM em software, como a TinySTM, não possuem recursos para realizar este monitoramento de forma automática. Elas exigem que o programador utilize instruções específicas de acesso à Memória Transacional em operações de leitura e escrita. Ou seja, a abstração de TM em realizar o monitoramento dos acessos à memória, de fato, é oferecida na forma de um novo conjunto de instruções na qual o endereço do dado a ser transacionado é indicado de forma explícita. A interface proposta possui uma abordagem diferente: o programador informa no início do bloco transacional quais são as variáveis a serem manipuladas em transações e, em um processo de tratamento do código, por compilação, por exemplo, as devidas instruções para acesso aos dados na Memória Transacional são inseridos. Pragmaticamente, não há diferença com TinySTM, no entanto, considera-se o nível de abstração proposto mais alto e, sem dúvida, mais próximo ao OpenMP. Soma-se ainda a possibilidade, na interface proposta, de indicar ainda os modos de acesso (R/W/RW) aos dados, visando atender critérios de otimização de execução que possam ser suportados pelas ferramentas de TM utilizadas.

5 EXPERIMENTAÇÃO E ANÁLISE DE DESEMPENHO

Neste capítulo são apresentados resultados de desempenho obtidos com a prototipação da interface proposta no Capítulo 4. Embora o desempenho seja apresentado em termos de tempo de execução, neste trabalho, o objetivo dos casos de estudo é o de avaliar a viabilidade de realização da interface em uma ferramenta de programação real. Adicionalmente, os experimentos validam a implementação utilizando duas interfaces de programação para Memórias Transacionais, TinySTM e GCC-TM.

A experimentação foi realizada considerando três conjuntos de programas. O primeiro é representado pelas aplicações do benchmark Cowichan. Programas deste benchmark foram implementados para validar a interface. Os resultados de desempenho correpondentes são apresentados na Seção 5.2. O segundo conjunto de programas é composto por uma aplicação do benchmark STAMP (bayes) e um programa desenvolvido como um benchmark para OpenMP (kmeans). Estes resultados são apresentados na Seção 5.3. Por fim, na Seção 5.4, é apresentado o terceiro conjunto de programas, que são variações sobre uma implementação concorrente recursiva de Fibonacci.

Aponta-se que a experimentação restringe-se a comparar o desempenho dos casos de estudo prototipados com TinySTM e GCC-TM e com versões originais em OpenMP puro. Não foram encontradas ferramentas operacionais oferecendo implementando as propostas apresentadas no Capítulo 3.

5.1 Metodologia Básica de Experimentação

As execuções dos experimentos das etapas 1 e 2 foram realizados dois multiprocessadores com arquitetura NUMA: **hydra** (arquitetura Opteron com 64 cores, 4 nós, 120 GB RAM) e **tekoha** (arquitetura Xeon com 192 cores, 8 nós, 120 GB RAM). O experimento da etapa 3 foi conduzido na máquina **kiriri**, com processador i5 2 cores e 4 threads, com 8 GB de memória RAM. Em todos experimentos, às máquinas estiveram com uso dedicado.

O índice de desempenho coletado é o tempo de execução, apresentado em se-

gundos, médio obtido em pelo menos 30 execuções de cada caso. Na apresentação destas médias é destacado quando as amostras de cada caso aderem a uma distribuição normal ou não. Para tanto, foi efetuado o teste Kolmogorov-Smirnov com 95% de confiança. As comparações entre os casos utilizam testes estatísticos adequados para obter posicionar o desempenho entre dois casos. Quando amostras aderem a uma normal foi utilizado o T de Student, caso contrário foi utilizado o teste U de Mann-Whitney, ambos também com 95% de confiança.

Os estudos de caso obtidos refletem uma combinação entre as versões dos programas a serem comparados, diferentes suportes de execução, diferentes arquiteturas (nas etapas 1 e 2) e diferente número de threads no suporte do runtime de OpenMP.

5.2 Experimentação Etapa 1: Validação do Protótipo

A primeira etapa de experimentação teve por objetivo validar o protótipo e coletar informações preliminares sobre o desempenho das duas ferramentas de programação oferecendo suporte à Memória Transacional.

5.2.1 Implementação dos Problemas

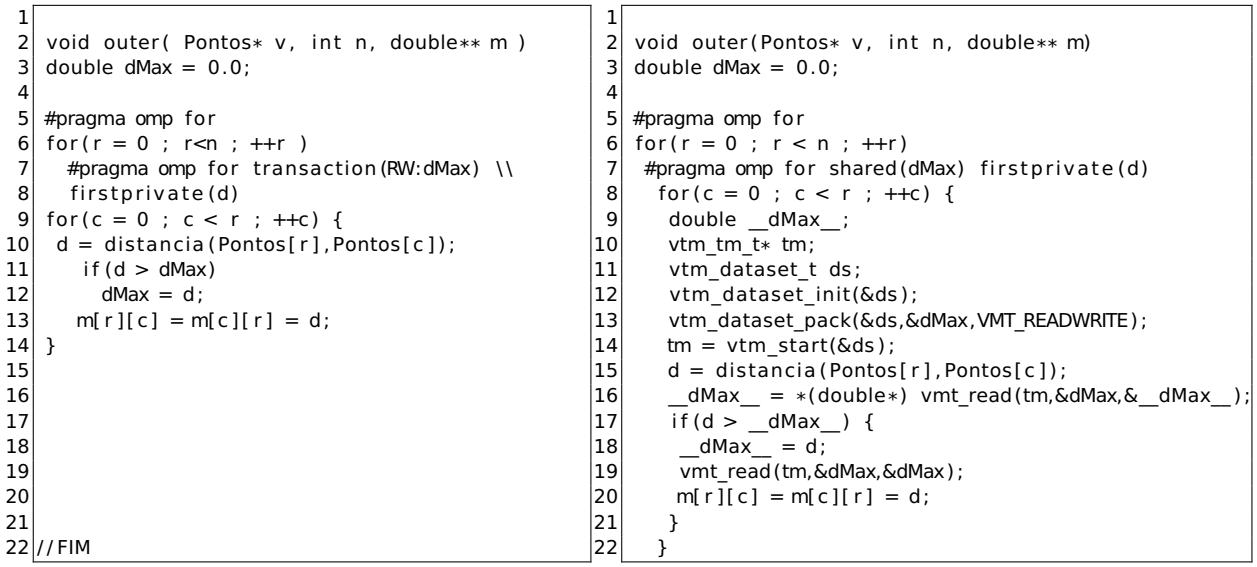
A validação da interface proposta se deu pela implementação dos seis programas do benchmark Cowichan (WILSON, 1994) em que foi identificada a aplicabilidade do modelo de Memória Transacional: hull, norm, outer, sor, thresh e vecdiff. O benchmark Cowichan não foi concebido para análise de desempenho, mas sim para caracterizar as ferramentas de programação paralela em termos da capacidade de representação de seus recursos de programação. Na Figura 22(a) é apresentado o código implementado na interface proposta para o problema `outer`, onde o uso da diretiva `transaction` é aplicada ao dado compartilhado `dMax`. O código na Figura 22(b) apresenta a implementação intermediária empregando *Vanilla*-TM.

A experimentação propriamente dita foi realizada substituindo as chamadas às primitivas de *Vanilla*-TM pelas chamadas aos serviços oferecidos pela TinySTM e por GCC-TM.

5.2.2 Coleta e Análise de Desempenho

Foram avaliadas três implementações para os problemas Cowichan selecionados, uma em OpenMP[1] e as outras duas obtidas a partir desta primeira versão pela sua adaptação à prototipação da interface proposta com *Vanilla*-TM versionada para o suporte de Memória Transacional oferecido por TinySTM e GCC-TM. Os experimen-

[1] Obtida em `https://code.google.com/archive/p/cowichan`, acesso em 9 de novembro de 2020.



```
1
2  void outer( Pontos* v, int n, double** m )
3  double dMax = 0.0;
4
5  #pragma omp for
6  for(r = 0 ; r<n ; ++r )
7    #pragma omp for transaction(RW:dMax) \\
8    firstprivate(d)
9  for(c = 0 ; c < r ; ++c) {
10  d = distancia(Pontos[r],Pontos[c]);
11    if(d > dMax)
12      dMax = d;
13   m[r][c] = m[c][r] = d;
14 }
15
16
17
18
19
20
21
22 //FIM
```

(a)

```
1
2  void outer(Pontos* v, int n, double** m)
3  double dMax = 0.0;
4
5  #pragma omp for
6  for(r = 0 ; r < n ; ++r)
7   #pragma omp for shared(dMax) firstprivate(d)
8    for(c = 0 ; c < r ; ++c) {
9     double __dMax__;
10    vtm_tm_t* tm;
11    vtm_dataset_t ds;
12    vtm_dataset_init(&ds);
13    vtm_dataset_pack(&ds,&dMax,VMT_READWRITE);
14    tm = vtm_start(&ds);
15    d = distancia(Pontos[r],Pontos[c]);
16    __dMax__ = *(double*) vmt_read(tm,&dMax,&__dMax__);
17    if(d > __dMax__) {
18     __dMax__ = d;
19     vmt_read(tm,&dMax,&dMax);
20     m[r][c] = m[c][r] = d;
21    }
22   }
```

(b)

Figura 22 – Código implementado na interface proposta para o problema `outer`: (a) *outer* Cowichan (b) *outer* `transaction`.

tos foram realizados em dois multiprocessadores com as arquitetura NUMA **hydra** e **tekoha**. Os experimentos consideraram três tamanhos de entrada do problema, definidos conforme a natureza de cada aplicação, identificados por: *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 fontes, bem como os scripts para execução e geração de gráficos, estão disponíveis em `https://github.com/adjardim/OpenMP_TM_TinySTM`. 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 e aplicando como teste de hipóteses o teste t de Student (significância de 95%), foram comparados os desempenhos das implementações com OpenMP, OpenMP/TinySTM, OpenMP/GCC-TM.

As tabelas 4, 5 e 6 exemplificam os resultados obtidos apresentando os tempos

Tabela 4 – Desempenho do caso: Entrada *pequena*, com 32 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,22	0,25	0,21	0,24	0,26	0,23
norm	0,06	0,15	4,76	0,04	0,08	7,07
outer	0,29	0,28	8,34	0,09	0,14	14,04
sor	4,25	0,28	0,26	0,07	0,09	0,07
thresh	1,13	16,13	34,60	0,14	30,93	51,19
vecdiff	0,1	0,25	22,36	0,03	0,17	39,2

de execução médios observados pela execução dos programas sobre entradas de tamanho pequeno, médio e grande com 32 threads no time de execução de OpenMP. O conjunto completo de dados coletados encontra-se no Apêndice A.

Tabela 5 – Desempenho do caso: Entrada *média*, com 32 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,8	0,94	0,81	0,74	0,73	0,74
norm	0,32	0,65	23,78	0,08	0,33	39,82
outer	0,47	0,68	33	0,39	0,54	59,38
sor	0,9	0,95	0,88	0,24	0,25	0,25
thresh	0,45	66,27	126,08	0,33	122,26	209,01
vecdiff	0,19	0,47	44,73	0,06	0,33	79,2

Tabela 6 – Desempenho do caso: Entrada *grande*, com 32 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	2,42	9,15	149,65	2,42	13,27	267,01
norm	0,61	1,28	47,54	0,15	0,63	80,79
outer	0,81	4,53	131,83	1,92	2,24	241,51
sor	4,01	4,14	4,51	0,79	0,9	1,06
thresh	1,55	265,14	502,84	0,87	490,6	1221,75
vecdiff	0,86	2,5	221,55	0,19	1,43	399,66

Nas tabelas apresentadas, as células marcadas em vermelho correspondem aos experimentos cujas amostras não aderiram a uma curva normal. Ou seja, estas médias não representam amostras cuja distribuição tenha se mostrado aderente a uma distribuição normal. Neste caso, os tempos médios são apresentados de forma ilustrativa, mas não foram utilizados na comparação dos desempenho entre as ferramentas. A discussão na sequência considera todo o conjunto das experimentos realizados, não apenas a ilustração dos resultados apresentada nas tabela apresentadas nesta seção.

Análise das amostras. O teste de normalidade das amostras permite compreender o impacto do tamanho do problema na representatividade dos tempos de execução coletados. Para problemas cujos tempos de execução são curtos, tendo como referência os tempos de execução com OpenMP, a variabilidade dos tempos

associados à gestão de todo o processo tem maior influência mais que o tempo efetivo da computação útil no tempo total. Este aspecto é ainda mais evidente com o decréscimo do desempenho em função do incremento de número de threads no time de execução de OpenMP. Isso pode ser constatado observando que, independente do número de threads utilizado, boa parte dos tempos coletados para entradas de tamanho pequeno, mesmo nas implementações com OpenMP, as amostras para cada caso de estudo não aderem a uma curva normal.

Execuções longas nas implementações com Memória Transacional também mostraram, em determinadas situações, alta variabilidade. Remetendo aos resultados obtidos, esta situação se apresenta nos tempos obtidos com GCC-TM nas aplicações hull e outer, para execuções com 32 e 64 threads tendo como entrada de tamanho grande. 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, não realizado neste trabalho, para identificar as opções de escalonamento de transações para os programas avaliados.

Teste de Hipóteses. Foram testadas duas hipóteses, a primeira que as execuções com OpenMP puro são mais eficientes que as oferecidas por TinySTM, comparando as médias de OpenMP com as oferecidas por TinySTM. A segunda hipótese é de que as implementações com TinySTM são mais eficientes que aquelas sobre GCC-TM. As tabelas 7, 8 e 9 ilustram a aplicação do teste t de Student para os dados apresentados nas tabelas 4, 5 e 6. É apresentado o $p - valor$ obtido em cada teste.

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 quase totalidade dos casos. A exceção é no programa sor quando submetido a uma entrada de tamanho grande. Este caso está destacado com células em vermelho na Tabela 9. Este caso ilustra uma situação onde não é possível afirmar a superioridade de OpenMP sobre TinySTM nem de TinySTM sobre GCC-TM. Neste programa, o teste aplicado aos resultados não foi capaz de confirmar, com 95% de confiança, que as médias obtidas pelas execuções com as três ferramentas pertencessem a populações distintas.

Tabela 7 – Resultado para $p - valor$ pelo teste t de Student para os dados da Tabela 4, para $p < 0,05$.

	OpenMP é melhor que TinySTM	**TinySTM é melhor que GCC-TM**
hull	0,00001	0,00001
norm	0,00001	0,00001
outer	0,00001	0,00001
sor	0,00192	0,007368
tresh	0,00001	0,00001
vecdiff	0,00001	0,00001

Tabela 8 – Resultado para $p - valor$ pelo teste t de Student para os dados da Tabela 5, para $p < 0,05$.

	OpenMP é melhor que TinySTM	TinySTM é melhor que GCC-TM
hull	0,00001	0,00001
norm	0,00001	0,00001
outer	0,00001	0,00001
sor	0,049146	0,049048
tresh	0,00001	0,00001
vecdiff	0,00001	0,00001

Tabela 9 – Resultado para $p - valor$ pelo teste t de Student para os dados da Tabela 6, para $p < 0,05$.

	OpenMP é melhor que TinySTM	TinySTM é melhor que GCC-TM
hull	0,00001	0,00001
norm	0,00001	0,00001
outer	0,00001	0,00001
sor	0,074099	0,101948
tresh	0,00001	0,00001
vecdiff	0,00001	0,00001

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 as diferenças 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 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.

No entanto, a quantidade de experimentos que retornaram médias representativas permitiu embasar as conclusões apresentadas sobre o desempenho.

5.3 Experimentação Etapa 2: Análise do Comportamento do Protótipo

Na segunda etapa de experimentação, dois programas desenvolvidos originariamente para análise de desempenho foram reimplementados com a interface proposta e tiveram seus resultados de desempenho avaliados. Um destes programas,

bayes, pertence ao benchmark STAMP (MINH et al., 2008), desenvolvido para avaliar o desempenho de ferramentas para programação com Memória Transacional. O segundo programa foi desenvolvido como benchmark para OpenMP[2]. O objetivo desta segunda etapa de experimentação é o de posicionar a solução proposta em relação a sua aplicabilidade em problemas concebidos dentro do contexto da programação com Memória Transacional e problemas concebidos para serem aplicados sobre OpenMP puro.

5.3.1 Bayes

Este programa, no benchmark STAMP (MINH et al., 2008), implementa um algoritmo para aprendizado de redes bayesianas. A complexidade do programa está associada ao número das variáveis consideradas e a estrutura de conexão entre estas variáveis. Estas conexões representam, em termos de probabilidade, a dependência condicional entre duas variáveis. O lançamento do programa permite a parametrização do tamanho do problema indicando o número de variáveis a serem tratadas e o grau de conectividade entre as variáveis.

Na implementação original deste problema[3] é calculada a função de verossimilhança para cada variável e, então, acumulado os valores obtidos para cada variável. A operação de cálculo é realizada de forma concorrente e a operação de acumulação para obtenção do valor global utiliza algum mecanismo de sincronização. A estrutura de código original, em OpenMP, é ilustrada na Figura 23, destacando partes do código relevantes para compreensão da modificação realizada para contemplar a interface proposta. Neste código, as primitivas prefixadas por `TM_` correspondem a macros implementadas no benchmark STAMP para permitir adaptação a diferentes interfaces para suporte à Memória Transacional. O código apresentado implementa parte do algoritmo de aprendizado da estrutura Bayesiana a partir dos dados de entrada, trecho este, paralelizado.

No código da Figura 23, a região paralela instancia tantas tarefas para executar o bloco de comandos apresentado quanto forem o número de threads no time de execução do *runtime* de OpenMP. Cada tarefa tem como entrada `learnerPtr`, onde encontram-se o conjunto de variáveis a ser tratado e também onde deve ser atualizado o resultado da computação. Em um primeiro momento, cada tarefa identifica o conjunto de variáveis que irá tratar e então acumula o resultado parcial da computação no resultado acumulado de sua partição (`baseLogLikelihood`). O resultado global, produzido pelo conjunto de tarefas (`learnerPtr->baseLogLikelihood`), corresponde ao somatório de todos os valores parciais computados. No programa

[2] O código utilizado como base da implementação encontra-se disponível em `https://github.com/manshi10/kmeans` (último acesso em 12/nov/2020).

[3] A implementação deste benchmark utilizada neste trabalho foi retirada do repositório de um de seus autores: `https://github.com/kozyraki/stamp`, com acesso em 7 de setembro de 2020.

```
1   #pragma omp parallel shared(learnerPtr, numVar) \
2                        private(myId,numThr,localBaseLogLikelihoods,\
3                        baseLogLikelihood,globalBaseLogLikelihood)
4   {
5    myId    = omp_get_thread_num();
6    numThr =  omp_get_num_thread();
7
8    createPartition(0, numVar, myId, numThr, &v_start, &v_stop);
9
10   for( v = v_start, v < v_stop ; ++v ) {
11    localBaseLogLikelihoods[v] += compute(... v ... );
12    baseLogLikelihood += localBaseLogLikelihood[v];
13   }
14   TM_BEGIN();
15   globalBaseLogLikelihood = TM_SHARED_READ_F(learnerPtr->baseLogLikelihood);
16   TM_SHARED_WRITE_F(learnerPtr->baseLogLikelihood,
17                     (baseLogLikelihood + globalBaseLogLikelihood));
18   TM_END();
19     ...
20  }
```

Figura 23 – Código esquemático do programa bayes
no benchmark STAMP (fragmento).

original, o somatório dos valores parciais para obtenção do valor global é realizado em uma transação, conforme identificado no trecho de código entre as primitivas `TM_BEGIN` e `TM_END`.

A adaptação deste programa para a interface proposta é apresentada na Figura 24. O código correspondente com a linguagem intermediária *Vanilla*-TM é apresentado na Figura 25. A prototipação com a interface proposta no programa introduziu, portanto, a instanciação de uma nova tarefa, exclusiva para manipulação da transação sobre o dado compartilhado.

```
1  #pragma omp task if(0) transaction(RW:learnerPtr->baseLogLikelihood);
2   {
3    globalBaseLogLikelihood = learnerPtr->baseLogLikelihood;
4    learnerPtr->baseLogLikelihood = baseLogLikelihood + globalBaseLogLikelihood;
5   }
6  #pragma omp taskwait
7  }
```

Figura 24 – Trecho de código no programa bayes alterado para a interface proposta.

5.3.2 K-means

A implementação utilizada pertence a um benchmark[4] desenvolvido para avaliar diferentes implementações do algoritmo kmeans em diferentes ferramentas de pro-

[4] A implementação original do benchmark encontra-se disponível em `http://users.eecs.northwestern.edu/wkliao/Kmeans/index.html`, último acesso em janeiro de 2021.

```
1           #pragma omp task if(0)
2            {
3              vtm_t dtas;
4              vtm_dataset_init(&dtas);
5              vtm_dataset_pack(&dtas,RW,learnerPtr->baseLogLikelihood);
6              vtm_start(dtas);
7              vtm_read(dtas,0,&globalBaseLogLikelihood);
8              tmp = baseLogLikelihood + globalBaseLogLikelihood;
9              vtm_write(dtas,0,&tmp);
10             vtm_commit(dtas);
11           }
12          #pragma omp taskwait
13          }
```

Figura 25 – Trecho de código no programa bayes com a linguagem intermediária *Vanilla*-TM.

gramação. Dentre as variações apresentadas na versão original deste benchmark, encontra-se uma implementação em OpenMP utilizando operações atômicas para manipulação de dados compartilhados. Esta implementação foi a utilizada para fins de avaliação no presente trabalho. Um trecho do código correspondente encontra-se na Figura 26. As operações atômicas em questão encontram-se nas linhas 9 e 12.

```
1  #pragma omp for private(i,j,index), firstprivate(numObjs,numClusters, \
2           shared(objects,clusters,membership,newClusters,newClusterSize), \
3           schedule(static), reduction(+:delta)
4   for (i=0; i<numObjs; i++) {
5      index = find_nearest_cluster(numClusters, numCoords, objects[i], clusters);
6      if (membership[i] != index) delta += 1.0;
7          membership[i] = index;
8      #pragma omp atomic
9      newClusterSize[index]++;
10     for (j=0; j<numCoords; j++)
11        #pragma omp atomic
12    newClusters[index][j] += objects[i][j];
13  }
```

Figura 26 – Trecho de código no programa kmeans original.

O trecho de código alternativo, produzido com a interface proposta, é apresentado na Figura 27 e seu correspondente em *Vanilla*-TM na Figura 28. As operações transacionais substituem aquelas realizadas de forma atômica.

5.3.3 Coleta e Análise de Desempenho

As execuções destes programas foram realizadas em duas arquiteturas multiprocessadas NUMA **hydra** e **tekoha**. Em ambos os problemas foram considerados três tamanhos de problema, pequeno, médio e grande. Os resultados, representados em tempos médios de execução, para 30 execuções, encontram-se na Tabela 10. Os programas identificados com -VAN são as implementadas com a interface

```
1  #pragma omp parallel transaction
2   {
3    #pragma omp for private(i,j,index),    firstprivate(numObjs,numClusters, \
4            shared(objects,clusters,membership,newClusters,newClusterSize), \
5            schedule(static), reduction(+:delta)
6     for (i=0; i<numObjs; i++) {
7      index = find_nearest_cluster(numClusters, numCoords, objects[i], clusters);
8      if (membership[i] != index) delta += 1.0;
9      membership[i] = index;
10     #pragma omp task if(0) transaction(RW:newClusterSize[index:1])
11      newClusterSize[index]++;
12     for (j=0; j<numCoords; j++)
13     #pragma omp task if(0)    transaction(RW:newCluster[index+j:1])
14      newClusters[index][j] += objects[i][j];
15   }
16  }
```

Figura 27 – Trecho de código no programa kmeans com a interface proposta.

proposta. O programa kmeans é o benchmark original implementado em OpenMP; o programa bayes é a versão disponível no benchmark STAMP, também implementado em OpenMP, com suporte da TinySTM em suas configurações padrão. São apresentadas as médias de tempo coletadas para execuções suportadas com OpenMP configurado com 2, 3, 8, 16 e 64 threads em cada arquitetura. Na tabela, as células com fundo vermelho indicam amostras que não aderiram a uma curva normal.

Tabela 10 – Comparativo dos tempos de execução: bayes e kmeans

	Hydra						Tekoha					
	2	4	8	16	32	64	2	4	8	16	32	64
						Pequeno						
kmeans	19,40	10,10	5,29	2,92	1,86	1,62	10,78	7,40	5,05	3,97	3,96	3,89
kmeans-VAN	32,91	38,65	83,84	167,36	401,66	534,04	37,24	50,40	117,27	223,94	459,53	918,57
bayes	10,60	10,84	12,91	11,04	11,86	11,42	5,45	5,76	6,73	5,97	5,29	5,71
bayes-VAN	6,76	3,66	5,70	6,41	3,77	4,10	3,55	2,24	2,81	3,40	2,96	2,17
						Médio						
kmeans	37,12	19,34	10,05	5,56	3,51	3,03	20,70	14,38	10,31	8,16	7,76	7,54
kmeans-VAN	23,92	88,25	152,68	372,36	846,79	1011,78	56,19	113,27	224,95	480,41	930,51	1842,23
bayes	14,12	18,21	12,36	15,67	19,99	16,46	7,33	8,73	9,46	7,04	7,55	8,46
bayes-VAN	8,57	5,05	16,38	9,98	9,12	8,93	3,60	4,77	7,66	4,72	3,23	7,02
						Grande						
kmeans	75,16	39,33	20,67	11,10	6,89	6,00	40,86	29,29	21,72	17,30	15,97	15,12
kmeans-VAN	147,07	221,28	335,41	761,71	1677,89	2152,70	126,21	247,09	485,72	1031,31	1872,23	3570,21
bayes	14,50	15,99	13,34	14,18	14,35	13,68	8,30	6,63	8,80	8,80	7,43	7,55
bayes-VAN	7,41	7,64	4,64	8,34	4,50	5,03	5,28	4,46	2,18	3,93	2,70	2,90

A metodologia do experimento envolveu a amostragem dos tempos de execução de cada uma das implantações. A versão do código implementando a interface proposta teve suporte da biblioteca TinySTM, configurado com suas opções padrão, para manipulação da de dados em Memória Transacional.

Análise das amostras. Neste experimento, observou-se que as amostras aderiram a uma distribuição normal em um maior número de casos. Na máquina **tekoha**, talvez devido a esta máquina oferecer um maior poder de processamento, cujo custo de gestão não é compensado pelo tamanho dos problemas explorados. Esta afirmação é apresentada verificando que naqueles casos onde o custo de processamento do problema é pequeno em relação à capacidade de paralelismo explorado, o tempo

```
1  #pragma omp parallel
2  {
3  #pragma omp for private(i,j,index), firstprivate(numObjs,numClusters, \
4            shared(objects,clusters,membership,newClusters,newClusterSize), \
5            schedule(static), reduction(+:delta)
6   for (i=0; i<numObjs; i++) {
7      index = find_nearest_cluster(numClusters, numCoords, objects[i], clusters);
8    if (membership[i] != index) delta += 1.0;
9    membership[i] = index;
10  #pragma omp task if(0)
11   {
12   vtm_dataset_init(&dtas );
13   vtm_dataset_pack(&dtas, RW, newClusterSize[index]);
14   vtm_start(dtas);
15   vtm_write(dtas, newClusterSize[index],  newClusterSize[index]+1);
16   vtm_commit(dtas);
17   }
18   for (j=0; j<numCoords; j++)
19  #pragma omp task if(0)
20   {
21   vtm_dataset_init(&dtas );
22   vtm_dataset_pack(&dtas, RW, newClusterSize[index]);
23   vtm_start(dtas);
24   vtm_write(dtas, newClusters[index][j],  newClusters[index][j]+objects[i][j]);
25   vtm_commit(dtas);
26   }
27  }
28 }
```

Figura 28 – Trecho de código no programa kmeans com a linguagem intermediária *Vanilla*-TM.

de execução na **tekoha** é mais alto. Da análise das amostras, verifica-se o indicativo de que a versão bayes-VAN apresenta ganho de desempenho frente à sua versão original.

Teste de Hipóteses. A primeira hipótese testada foi verificar se a implementação kmeans-VAN introduz sobrecusto frente a versão OpenMP puro para confirmar a observação dos dados amostrais. Foi constatado que, com 95% de confiança, existe diferença entre as médias dos tempos coletados nestes dois programas em todos os casos considerados. Na verdade, em termos absolutos, esta diferença de desempenho é bastante significativa, aumentando com o acréscimo de threads no suporte de execução. O código original, em OpenMP, utiliza operações atômicas, suportadas diretamente pela arquitetura, para acessar os dados compartilhados. Já na solução proposta, é necessária a criação de uma nova tarefa para executar a operação na sua forma transacional, incorrendo, neste ponto, introdução de sobrecustos significativos.

A segunda hipótese considerada foi verificar se existe ganho na fase *learning* do programa bayes utilizando a versão com a interface proposta. Foi constatado, com

95% de confiança, que as amostras proveem de populações distintas, indicando ganho de desempenho efetivo. Uma única exceção: não é possível afirmar com o nível de confiança desejado (95%), que no caso em que o conjunto de dados de entrada foi médio e executado com 8 threads existe diferença nas amostras. A questão que se coloca é de onde provem este ganho de desempenho, considerando que a versão original do programa também é executada em OpenMP com suporte da TinySTM. Ainda que novos experimentos devam ser realizados para avaliar com maior profundidade detalhes do comportamento da execução, observa-se que a diferença entre as implementações está na necessidade de, na implementação com a interface proposta, do lançamento de uma nova tarefa para executar as operações sobre a Memória Transacional. As decisões de escalonamento de OpenMP podem ter privilegiado a execução de tais tarefas, reduzindo a ocorrência de conflitos nos mecanismos internos de sincronização da biblioteca TinySTM.

A terceira hipótese foi verificar se existe diferença de desempenho na execução de bayes com tamanhos diferentes de entrada. Considerando os tamanhos pequeno, médio e grande, considerando dois threads no time de execução, com 95% de confiança os desempenhos são diferentes nas quatro implementações construídas. Esta informação indica que o tamanho do problema influencia o desempenho do programa.

A quarta hipótese foi avaliar se na aplicação bayes, o número de threads influencia no desempenho do programa. Foram testados, com 95% de confiança, os desempenhos obtidos para cada implementação, em cada máquina, considerando o tempo de execução com 2 e 4 threads, 4 e 8, 8 e 16, 16 e 32 e 32 e 64. Havendo duas máquinas, duas versões do programa, 3 tamanhos de entrada e 5 pares de threads, o número de combinações possíveis é de 60. Destes, não foi possível identificar, nos testes aplicados, diferença nas amostras obtidas em 33 casos. Os resultados, portanto não permitiram constatar que o número de threads influencia no desempenho dos casos analisados.

5.4 Etapa 3: Recursividade

O estudo sobre o comportamento, em tempo de execução, de programas implementando algoritmos recursivos com a interface proposta foi baseado na análise de diferentes implementações do algoritmo de Fibonacci. Este estudo de caso foi concebido para avaliar o uso da interface proposta em um algoritmo concorrente em que ocorre aninhamento na criação de tarefas.

Para avaliação deste estudo de caso, foram implementadas cinco versões deste programa, duas em OpenMP, uma em TinySTM e outras duas em Vanilla. Sendo o objetivo do experimento analisar o comportamento de execução do suporte provido

por Vanilla, e o sobrecusto inserido. As implementações em OpenMP possuem uma estrutura similar a utilizada nas implementações com suporte à Memória Transacional. Neste caso, foi utilizado uma seção crítica para obtenção do resultado final.

```
1
2  void fib( int n, int *r ) {
3    if (n<2) {
4      #pragma omp critical
5        *r += n;
6    }
7    else {
8      #pragma omp task
9        fib(n-1,r);
10     #pragma omp task
11       fib(n-2,r);
12     #pragma omp taskwait
13   }
14 }
15
16
17
18 //FIM
```

(a)

```
1
2  void outer(Pontos* v, int n, double** m)
3  double dMax = 0.0;
4
5  void fib( int n, int *r ) {
6    if( n<2 ) {
7      #pragma omp task if(0) transaction(A:r)
8        *r += n;
9      #pragma omp taskwait
10   }
11   else {
12     #pragma omp task transaction(D:r)
13       fib(n-1,r);
14     #pragma omp task transaction(D:r)
15       fib(n-2,r);
16     #pragma omp taskwait
17   }
18 }
```

(b)

Figura 29 – Casos de Estudo para algoritmo recursivo. (a) **SOMP**: Implementação em OpenMP. (b) **IFVanilla**: Implementação na interface proposta.

A Figura 29 apresenta duas versões dos programas implementados. Em Figura 29.(a) a versão OpenMP, em Figura 29.(b) a versão com a interface proposta. As cinco versões construídas são as seguintes:

SOMP : Versão clássica do programa em OpenMP, apresentado em 29.(a), onde o endereço de uma variável é recursivamente submetido às tarefas descendentes para, na tarefa folha, haver a contribuição ao cálculo, sendo observado acesso ao dado compartilhado em uma seção crítica.

TGOMP : Versão OpenMP utilizando a cláusula `task_reduction` para acumular, nas folhas, os resultados parciais do cálculo computado pelas diferentes tarefas.

TinySTM : Implementação em OpenMP utilizando a biblioteca TinySTM para, nas tarefas folhas, acumular a contribuição da tarefa no cálculo final. O endereço da variável compartilhada é, recursivamente, passado a cada nível de recursão instanciado.

Vanilla : Implementação em OpenMP utilizando a interface proposta, transcrita com a biblioteca *Vanilla*-TM tendo suporte de TinySTM. Nesta implementação, também o endereço da variável compartilhado é passado recursivamente a cada nível de recursão. Ao atingir a base desta árvore de recursão, a tarefa folha cria uma nova tarefa responsável por realizar o acesso transacionado sobre o dado compartilhado.

IFVanilla : Apresentada na Figura 29.(b), a tarefa criada para realizar a transação é instanciada com a cláusula `if(0)`, indicando que a tarefa deve ser executada imediatamente.

A versão do código **IFVanilla** consiste em uma otimização viável do código **Vanilla** que pode ser detectada pelo uso do modificador de acesso `adopt` em uma diretiva `task`. A implementação desta etapa no processo tradução (Seção 4.5.3 deve considerar que a tarefa instanciada irá manipular ela própria a transação, então o custo de criação de uma nova tarefa pode ser suprimido, priorizando sua execução em profundidade.

Na Tabela 11 são apresentados os tempos de execução obtidos nas diferentes versões do programa em uma máquina com processador Intel I5, com 2 *cores* físicos e 4 threads, e 8 GB de RAM. Os tempos, são apresentados em segundos, correspondendo a uma média de 30 execuções. Os tempos marcados em células vermelhas correspondem a médias cuja amostra do tempo não aderiu a uma distribuição normal com 95% de confiança. Nos demais casos, houve aderência. As medidas apresentadas correspondem aos tempos coletados para obtenção das posições 30, 35 e 40 da série de Fibonacci, com o runtime de OpenMP configurado com 1, 2 e 4 threads. Observa-se que as versões do programa são apresentadas na ordem de desempenho, sendo a primeira versão aquela que apresentou melhor desempenho. A diferença de desempenho, e esta classificação, são baseados nos resultados do teste T de Student e do teste U de Mann-Whitney, para os casos de amostras com distribuição normal ou não, respectivamente. Nestes testes, os resultados mostraram que, na maioria dos casos, com 95% de confiança, os resultados de desempenho são distintos. Exceções observadas: não é possível afirmar que as versões SOMP e TinySTM para cálculo de Fibonacci de 30 e 35, com, respectivamente 4 e 2 threads no runtime sejam diferentes.

Tabela 11 – Estudo de caso com algoritmo recursivo.
Tempo em segundos

Versão	Threads	Fibo(n)		
		30	35	40
SOMP	1	0,27	2,90	31,84
	2	1,09	12,17	133,87
	4	1,10	12,19	134,45
TinySTM	1	0,28	3,10	34,28
	2	1,10	12,27	136,01
	4	1,66	12,83	141,47
IFVanilla	1	0,32	3,55	39,39
	2	1,15	12,68	140,82
	4	1,19	13,23	146,94
Vanilla	1	0,42	4,68	51,04
	2	1,68	13,40	205,37
	4	1,75	19,48	216,14
TGOMP	1	0,55	6,10	67,36
	2	1,30	14,37	159,87
	4	1,28	13,90	157,03

É possível observar, na tabela, que o custo computacional, refletido no tempo de execução, cresce ao aumentar o número de threads. Isso decorre do custo computacional do problema ser pequeno, em função do sobrecusto de execução. No entanto, os resultados apresentados permitem analisar o custo adicionado pela interface proposta sobre as alternativas de implementação em OpenMP puro e diretamente em TinySTM. A Tabela 12 expõe, na forma de diferença percentual, o sobrecusto do suporte da versão IFVanilla para as implementações em OpenMP puro (SOMP) e diretamente com a TinySTM. Os resultados permitem supor uma estabilidade no sobrecusto em função do número de threads no runtime, independente do tamanho do problema considerado. Uma exceção, um ganho de desempenho de 28,31% no caso do cálculo de Fibonacci de 30 com 4 threads. Este caso é visto como uma exceção pois os valores observados neste contexto são bastante pequenos e, por consequência, muito sujeitos a variações de estado de elementos externos, embora medidas tenham sido tomadas para execução do experimento e certificação dos dados colhidos.

Tabela 12 – Sobrecustos do suporte à interface proposta.

		Fibo(n)					
		SOMP			**TinySTM**		
		30	35	40	30	35	40
IFVanilla	1	18,52%	22,41%	23,71%	14,29%	14,52%	14,91%
	2	5,50%	4,19%	5,19%	4,55%	3,34%	3,54%
	4	8,18%	8,53%	9,29%	-28,31%	3,12%	3,87%

Por fim, como último comentário a este experimento, aponta-se a versão IFVanilla apresenta desempenho superior a versão Vanilla. A explicação deste comportamento está nas próprias decisões de escalonamento de OpenMP. A versão IFVanilla instancia a tarefa para contabilizar contribuições ao cálculo da posição da série utilizando a cláusula `if(0)`, o que implica em imediata execução da nova tarefa, reduzindo os sobrecustos do ambiente OpenMP em gerar uma nova tarefa cujo custo computacional é muito pequeno.

5.5 Discussão Sobre os Resultados

Este capítulo foi dedicado à avaliação de desempenho da interface proposta em diferentes casos de estudo. Mesmo sendo a proposta e a validação da extensão para uso de Memória Transacional sobre OpenMP o objetivo principal deste trabalho, a validação de desempenho é importante por atestar a capacidade de implementação do modelo e posicionar a performance potencial da solução entre outras opções disponíveis para programação com Memória Transacional sobre OpenMP. O capítulo iniciou apresentando a instanciação do protótipo de experimentação e então conduziu a avaliação de desempenho em três etapas. Primeiro foi avaliado a operacionalidade do protótipo em si e, então, aferido desempenho com dois benchmarks. A

terceira etapa de experimentação avaliou o uso da interface proposta em algoritmos recursivos.

O protótipo foi construído com apoio da linguagem intermediária *Vanilla*-TM. Esta linguagem intermediária foi desenvolvida no contexto do trabalho para permitir substituir o suporte STM sem a necessidade de alteração no código da aplicação. As implementações realizadas contemplaram o uso de dois suportes: TinySTM e GCC-TM. Algumas funcionalidades previstas em *Vanilla*-TM, como a possibilidade de indicar o modo de acesso aos dados e configurar as políticas de gestão da Memória Transacional não são, atualmente, oferecidas pelos suportes utilizados. Não foram, portanto, devidamente avaliados mas indica que *Vanilla*-TM poderá ser adaptada a outras ferramentas.

A primeira etapa de experimentação foi a validação do protótipo. Para tal, um conjunto de programas do benchmark Cowichan foi implementado. O desempenho obtido com o protótipo, sendo suportado por TinySTM e por GCC-TM, foi comparado com o fornecido pela implementação original OpenMP. O benchmark Cowichan foi desenvolvido especialmente para avaliar o poder de expressão de ferramentas para programação concorrente e paralela, não sendo, portanto, voltado a análise de desempenho efetivamente. No entanto, esta etapa de experimentação foi válida por atestar a operacionalidade do protótipo e indicar que o suporte oferecido por TinySTM é mais eficiente que o oferecido por GCC-TM.

A segunda etapa de experimentação avaliou ou uso da interface proposta em dois benchmarks de naturezas distintas. Um destes benchmarks consistiu em um programa originalmente escrito em OpenMP (kmeans) para avaliação de desempenho, que foi versionado para incorporar a interface proposta. O segundo benchmark (bayes) foi desenvolvido especificamente para avaliar desempenho de interfaces para programação com Memória Transacional em software. O simples versionamento destas aplicações para uso da interface proposta mostrou um desempenho bastante inferior no benchmark para desempenho em OpenMP e desempenho ligeiramente superior no benchmark para ferramentas de programação com Memória Transacional. Estes resultados abrem perspectivas de trabalhos futuros, em particular, como articular o novo recurso de programação no desenvolvimento de novos programas.

A terceira etapa de experimentação avaliou o uso da interface proposta em algoritmos recursivos. A implementação de diferentes versões do problema permitiram verificar que a interface proposta, com ajuda das propriedades `adopt` e `defer`, permite escrever um programa em que não ocorre aninhamento de transações. Também foi identificado um ligeiro sobrecusto em relação ao uso direto de TinySTM sobre OpenMP. Este resultado indica que trabalhos futuros, envolvendo o suporte nativo à Memória Transacional em OpenMP pode ser interessante em termos de desempenho.

6 CONCLUSÃO

As dificuldades de desenvolvimento de programa, e também manutenção, são muitas. Centrando nas questões relacionadas à programação multithreaded, na literatura surgem propostas de recurso de programação com mais alto nível de abstração, facilitando o processo de desenvolvimento e manutenção de código. O uso do modelo de TM foi proposto com este intuito.

Embora a literatura apresente grandes benefícios do modelo de Memória Transacional, sua incorporação em ferramentas de programação multithreaded em uso no mercado não é uma realidade. Neste trabalho foi proposta uma extensão à OpenMP para suporte à TM.

O principal resultado foi a incorporação do mecanismo de Memória Transacional em OpenMP e sua validação, realizada com apoio de um protótipo desenvolvido sobre ferramentas oferecendo suporte à TM, TinySTM e GCC-TM.

As etapas seguintes do trabalho foram a definição da interface e a experimentação e análise de desempenho. Uma vez definida a interface, ela foi validada implementando programas de um benchmark – Cowichan – concebido para avaliar o poder de expressão das linguagens de programação paralela. Uma avaliação baseada em GQM permitiu comparar a programabilidade da interface proposta com as oferecidas por trabalhos que também propuseram extensões da interface de programação de ferramentas de programação multithread para contemplar o uso de Memória Transacional, e mesmo com OpenMP puro. Os dados resultantes da análise dos códigos mostram, graças a expressividade do recurso proposto, que o uso da cláusula `transaction` proporcionou uma redução no número de linhas, diretivas e invocações de diretivas em comparações com o código OpenMP original e com as versões empregando as extensões OpenMP propostas nos trabalhos relacionados. Na experimentação e análise de desempenho, o objetivo dos casos de estudo foi o de avaliar a viabilidade de realização da interface em uma ferramenta de programação real. A validação do protótipo foi realizada pela execução e avaliação de três implementações para problemas Cowichan selecionados. Uma em OpenMP e as outras duas obtidas a partir desta primeira versão, pela sua adaptação à prototi-

pação da interface proposta com *Vanilla*-TM versionada para o suporte de Memória Transacional oferecido por TinySTM e GCC-TM, duas interfaces de programação para Memórias Transacionais. Foi possível verificar que o protótipo foi operacional tanto com TinySTM como com GCC-TM, embora a comparação sobre os tempos de desempenho coletados não tenham permitido ver vantagens em relação ao desempenho do protótipo frente a opção de execução sobre OpenMP puro. Sobre isso, dois pontos relevantes: o benchmark foi concebido para avaliar o poder de expressão, não desempenho; a implementação foi concebida para uma solução em OpenMP puro e então adaptada para a interface com TM, não sendo inicialmente pensada para ser com TM.

A prototipação da extensão proposta explorou a *Vanilla*-TM, uma interface que consiste em uma representação intermediária concebida com o objetivo de permitir que a extensão proposta possa ser suportada por diferentes ferramentas de suporte a Memória Transacional em Software. Entre as ferramentas com suporte ao modelo de TM em Software, esta tese apontou TinySTM e GCC-TM como alternativas à implementação de *Vanilla*-TM. TinySTM representa a implementação de TM sob a forma de uma biblioteca, compatível com programas C/C++. GCC-TM representa o suporte no compilador Gnu para a linguagem C/C++, para a especificação de Memórias Transacionais desta linguagem. A utilização da cláusula operacional ao conjunto de serviços OpenMP `transaction`, com apoio de *Vanilla*-TM, é suportada pela ferramenta de STM selecionada, sem interferência no modo de operação de OpenMP e sem utilizar nenhuma facilidade oferecida por este padrão. Uma avaliação de desempenho foi realizada implementando um programa de benchmark para TM (Bayes) e outro para OpenMP (kmeans), reimplementados com a interface proposta e tiveram seus resultados de desempenho avaliados, com o objetivo de posicionar a solução proposta em relação a sua aplicabilidade em problemas concebidos dentro do contexto da programação com Memória Transacional e problemas concebidos para serem aplicados sobre OpenMP puro. Além disso, foi realizado um experimento que teve como objetivo analisar o comportamento de execução do suporte provido por Vanilla, e o sobrecusto inserido. Nesta análise foram implementados algoritmos recursivos, com cinco versões do algoritmo de Fibonacci, duas em OpenMP, uma em TinySTM e outras duas em Vanilla. Quanto ao custo computacional, refletido no tempo de execução, ele cresce ao aumentar o número de threads. Isso acontece devido ao custo computacional do problema ser pequeno, em função do sobrecusto de execução. Os resultados apresentados permitiram analisar o custo adicionado pela interface proposta sobre as alternativas de implementação em OpenMP puro e diretamente em TinySTM, e permitiram supor uma estabilidade no sobrecusto em função do número de threads no runtime, independente do tamanho do problema considerado. Os resultados indicaram que a interface proposta apresenta um ligeiro sobrecusto

em relação ao uso direto de TinySTM sobre OpenMP.

6.1 Principais Contribuições

As contribuições científicas deste trabalho se deram no contexto da computação paralela, em interfaces para programação multithread e aplicação do conceito de Memória Transacional, especificamente. As contribuições são discutidas a seguir.

Avanço no estado da arte das interfaces para programação multithreaded com a inclusão de suporte ao modelo de Memória Transacional em Software. Nesta tese foi estendido o estado da arte em interfaces para programação concorrente multithreaded, tendo como estudo de caso a introdução de recursos para manipulação de Memórias Transacionais sobre OpenMP. O diferencial buscado no desenvolvimento do trabalho foi o de garantir que a ferramenta de programação no qual esta extensão foi introduzida – OpenMP – não tivesse suas características de programação alteradas e oferecesse o novo recurso de programação de forma consoante com os recursos já disponíveis nesta ferramenta.

Avanço na compreensão dos limites de desempenho na adoção de TM em interfaces de programação multithreaded. O foco da pesquisa foi buscar uma alternativa que permita aumentar a eficiência no uso dos recursos paralelos do hardware disponível, ao mesmo tempo que garanta níveis de produtividade no ciclo de desenvolvimento de código. Neste trabalho, foram realizados estudos sobre o desempenho de OpenMP utilizando suporte à manipulação de dados na forma transacional. A implementação deste suporte não foi realizado de forma nativa, mas sim, apoiado em ferramentas de suporte à Memória Transacional em software. As questões de desempenho não foram completamente tratadas, embora o conjunto de resultados obtidos pelos experimentos indiquem que a incorporação da interface proposta sobre OpenMP não adicionou sobrecustos consideráveis à utilização direta, em OpenMP, das ferramentas de Memória Transacional. Por outro lado, foi observado que em programas inicialmente desenvolvidos para OpenMP, a simples adaptação do algoritmo não trouxe resultados satisfatórios. Sendo este ponto considerado ainda aberto a novos experimento, o indicativo tomado é que o uso de Memória Transacional sobre OpenMP requer que as aplicações sejam desenvolvidas com algoritmos apropriados ao novo modelo de memória.

Implementação de programas do benchmark Cowichan com OpenMP e suporte de TM. Foram realizadas implementações dos códigos OpenMP originais dos problemas Cowichan, as versões empregando as extensões OpenMP propostas por trabalhos que também propuseram extensões semelhantes, e as versões com OpenMP utilizando os suportes de Memória Transacional em TinySTM e GCC-TM. A avaliação baseada em GQM e a análise dos códigos mostraram que o uso da cláusula

`transaction` proporcionou uma redução no número de linhas, diretivas e invocações de diretivas em comparações com o código OpenMP original e com as versões empregando as extensões OpenMP propostas nos trabalhos relacionados.

Apresentação de uma representação intermediária para construção de programas OpenMP com diferentes suportes de TM em Software. Foi apresentada a *Vanilla-TM*, uma representação intermediária para permitir a portabilidade das implementações realizadas entre diferentes ferramentas de programação com suporte à TM. A interface *Vanilla*-TM consiste em uma linguagem intermediária. Sua especificação tem como objetivo atender as necessidades de diferentes ferramentas oferecendo recursos para exploração de Memória Transacional em Software. Esta tese, entre as ferramentas com suporte ao modelo de TM em Software, apontou TinySTM e GCC-TM como alternativas à implementação de *Vanilla*-TM

Análise do desempenho de um benchmark escrito para OpenMP e outro para TM, e também de um algoritmo recursivo. Foi realizada uma avaliação de desempenho pela implementação de um programa de benchmark para TM (Bayes) e outro para OpenMP (kmeans), reimplementados com a interface proposta e tiveram seus resultados de desempenho avaliados. A análise de um estudo de caso de aplicação de Memória Transacional em um algoritmo recursivo implementado em OpenMP foi realizada pela implementação de algoritmos recursivos, com cinco versões do algoritmo de Fibonacci, duas em OpenMP, uma em TinySTM e outras duas em Vanilla, com o objetivo de analisar o comportamento de execução do suporte provido por Vanilla, e o sobrecusto inserido.

6.2 Trabalhos Futuros

Para trabalhos futuros, contempla-se especificar a interface Vanilla-TM de forma que ela seja representativa de um espectro maior de ferramentas STM e ainda contemplar espaço para sua extensibilidade.

Os resultados da segunda etapa de experimentação, envolvendo os benchmarks kmeans e bayes, também abrem perspectivas de trabalhos futuros, em particular, como articular o novo recurso de programação no desenvolvimento de novos programas. Os resultados da terceira etapa de experimentação indicaram que a interface proposta na tese apresentou um ligeiro sobrecusto em relação ao uso direto de TinySTM sobre OpenMP. Tais resultados indicam que trabalhos envolvendo o suporte nativo à Memória Transacional em OpenMP podem ser interessantes em termos de desempenho.

Além das possibilidades acima, visa-se realizar uma implementação da definição proposta, que permite a análise de características a nível de execução como escalabilidade, detecção de conflitos e sincronização. Características como aninhamento

de transações, definição de *read* e *write set*, chamadas de biblioteca e utilização de E/S também podem influenciar a definição da extensão para TM, portanto também serão levados em consideração. Além disso, a efetiva introdução da extensão em OpenMP e uma avaliação de desempenho considerando benchmark específico para avaliar ferramentas o modelo de Memória Transacional também são contemplados.

REFERÊNCIAS

ABDELKHALEK, A.; BILAS, A. Parallelization and performance of interactive multiplayer game servers. **18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.**, [S.l.], p.72–, 2004. (Cited on page 70.)

ADL-TABATABAI, A.-R. et al. Compiler and Runtime Support for Efficient Software Transactional Memory. **SIGPLAN Not.**, New York, NY, USA, v.41, n.6, p.26–37, June 2006. (Cited on page 18.)

AHMED, S.; BAGHERZADEH, M. What Do Concurrency Developers Ask about? A Large-Scale Study Using Stack Overflow. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, 12., 2018, New York, NY, USA. **Proceedings. . .** Association for Computing Machinery, 2018. (ESEM '18). (Cited on page 31.)

ANSARI, M. et al. Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory. In: ALGORITHMS AND ARCHITECTURES FOR PARALLEL PROCESSING, 2008, Berlin, Heidelberg. **Anais. . .** Springer Berlin Heidelberg, 2008. p.196–207. (Cited on page 70.)

ANVIK, J.; SCHAEFFER, J.; SZAFRON, D.; TAN, K. Asserting the Utility of CO2P3S Using the Cowichan Problem Set. **J. Parallel Distrib. Comput.**, Orlando, p.1542–1557, 2005. (Cited on page 73.)

BAEK, W. et al. The OpenTM Transactional Application Programming Interface. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURE AND COMPILATION TECHNIQUES, 16., 2007, Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2007. p.376–387. (PACT '07). (Cited on pages 18, 20, 65, 66, 67, 72, and 74.)

BAEZA-YATES, R. A.; RIBEIRO-NETO, B. **Modern Information Retrieval**. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. (Cited on page 18.)

Baugh, L.; Zilles, C. An Analysis of I/O And Syscalls In Critical Sections And Their Implications For Transactional Memory. In: ISPASS 2008 - IEEE INTERNATIONAL SYM-

POSIUM ON PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE, 2008. **Anais...** [S.l.: s.n.], 2008. p.54–62. (Cited on page 40.)

BERGER, E. D.; YANG, T.; LIU, T.; NOVARK, G. Grace: Safe Multithreaded Programming for C/C++. In: ACM SIGPLAN CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, 24., 2009, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2009. p.81–96. (OOPSLA '09). (Cited on page 40.)

Bienia, C.; Kumar, S.; Singh, J. P.; Li, K. The PARSEC benchmark suite: Characterization and architectural implications. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT), 2008., 2008. **Anais...** [S.l.: s.n.], 2008. p.72–81. (Cited on page 18.)

BIHARI, B. L. et al. A Case for Including Transactions in OpenMP II: Hardware Transactional Memory. In: OPENMP IN A HETEROGENEOUS WORLD, 2012, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2012. p.44–58. (Cited on pages 68 and 74.)

BLUMOFE, R. D. et al. Cilk: An Efficient Multithreaded Runtime System. **SIGPLAN Not.**, New York, NY, USA, v.30, n.8, p.207–216, Aug. 1995. (Cited on page 59.)

BOEHM, H. et al. Transactional Language Constructs for C++. **ISO/IEC JTC1/SC22 (Programming languages and operating systems) WG21 (C++)**, [S.l.], 2012. (Cited on page 37.)

BOEHM, H.-J. Threads cannot be implemented as a library. **ACM Sigplan Notices**, [S.l.], v.40, n.6, p.261–268, 2005. (Cited on page 39.)

BONNICHSEN, L.; PODOBAS, A. Using transactional memory to avoid blocking in OpenMP synchronization directives : Don't wait, speculate! In: INTERNATIONAL WORKSHOP ON OPENMP, IWOMP 2015 :, 11., 2015. **Anais...** [S.l.: s.n.], 2015. n.9342, p.149–161. (Lecture Notes in Computer Science). QC 20160203. (Cited on page 63.)

BOYD-WICKIZER, S. et al. An Analysis of Linux Scalability to Many Cores. In: USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, OSDI'10, 2010. **Anais...** [S.l.: s.n.], 2010. (Cited on page 71.)

BRESHEARS, C. **The Art of Concurrency**: A Thread Monkey's Guide to Writing Parallel Applications. [S.l.]: O'Reilly Media, Inc., 2009. (Cited on pages 27 and 28.)

BUDIMLIĆ, Z. et al. Concurrent Collections. **Sci. Program.**, Amsterdam, The Netherlands, The Netherlands, v.18, n.3-4, p.203–217, Aug. 2010. (Cited on page 60.)

BUSCH, C.; HERLIHY, M.; POPOVIC, M.; SHARMA, G. Fast Scheduling in Distributed Transactional Memory. In: ACM SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES, 29., 2017, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2017. p.173–182. (SPAA '17). (Cited on page 62.)

BUTENHOF, D. R. **Programming with POSIX Threads**. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. (Cited on page 18.)

CALDIERA, V. R. B. G.; ROMBACH, H. D. The goal question metric approach. **Encyclopedia of software engineering**, [S.l.], p.528–532, 1994. (Cited on page 81.)

CASCAVAL, C. et al. Software Transactional Memory: Why Is It Only a Research Toy? **Queue**, New York, NY, USA, v.6, n.5, p.46–58, Sept. 2008. (Cited on page 18.)

CASTRO, M. et al. **Analyzing Software Transactional Memory Applications by Tracing Transactions**. [S.l.: s.n.], 2010. Research Report. (Cited on page 56.)

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. (Ed.). **JAI**. Porto Alegre: Sociedade Brasileira de Computação, 2014. p.41–83. (Cited on pages 17, 51, and 59.)

CHANDRA, R. et al. **Parallel Programming in OpenMP**. San Francisco: Morgan Kaufmann, 2001. (Cited on pages 41 and 42.)

CHAPMAN, B.; JOST, G.; PAS, R. v. d. **Using OpenMP**: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). [S.l.]: The MIT Press, 2007. (Cited on pages 41, 42, and 45.)

Chi Cao Minh; JaeWoong Chung; Kozyrakis, C.; Olukotun, K. STAMP: Stanford Transactional Applications for Multi-Processing. In: IEEE INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION, 2008., 2008. **Anais...** [S.l.: s.n.], 2008. p.35–46. (Cited on page 62.)

DAGUM, L.; MENON, R. OpenMP: An Industry-Standard API for Shared-Memory Programming. **IEEE Comput. Sci. Eng.**, Los Alamitos, CA, USA, v.5, n.1, p.46–55, Jan. 1998. (Cited on pages 40 and 59.)

DIERBACH, C. Python as a first programming language. **Journal of Computing Sciences in Colleges**, [S.l.], v.29, n.3, p.73–73, 2014. (Cited on page 63.)

FELBER, P.; FETZER, C.; RIEGEL, T. Dynamic Performance Tuning of Word-based Software Transactional Memory. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 13., 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.237–246. (PPoPP '08). (Cited on pages 23, 55, and 56.)

FENG, M.; GUPTA, R.; NEAMTIU, I. Programming Support for Speculative Execution with Software Transactional Memory. In: IEEE 27TH INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING WORKSHOPS AND PHD FORUM, 2013., 2013, USA. **Proceedings...** IEEE Computer Society, 2013. p.394–403. (IPDPSW '13). (Cited on page 39.)

Free Software Foundation. **Transactional Memory in GCC**. Disponível em: <http://gcc.gnu.org/wiki/ TransactionalMemory>. (Cited on page 54.)

FüRLINGER, K.; KOWALEWSKI, R.; FUCHS, T.; LEHMANN, B. Investigating the Performance and Productivity of DASH Using the Cowichan Problems. In: PROC OF WORKSHOPS OF HPC ASIA, 2018, New York. **Anais...** ACM, 2018. p.11–20. (Cited on page 73.)

GAJINOV, V. et al. QuakeTM: Parallelizing a Complex Sequential Application Using Transactional Memory. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 23., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.126–135. (ICS '09). (Cited on page 70.)

GNU TM Library. **The GNU Transactional Memory Library**. Disponível em: <http://gcc.gnu.org/onlinedocs/libitm.pdf>. (Cited on pages 54 and 55.)

GOTTSCHLICH, J. E.; BOEHM, H.-J. Generic Programming Needs Transactional Memory. In: ACM, 2013. **Anais...** [S.l.: s.n.], 2013. (Cited on page 70.)

GUERRAOUI, R.; HERLIHY, M.; POCHON, B. Towards a Theory of Transactional Contention Managers. In: TWENTY-FIFTH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2006, New York, NY, USA. **Proceedings...** ACM, 2006. p.316–317. (PODC '06). (Cited on page 34.)

GUERRAOUI, R.; KAPALKA, M. **Principles of Transactional Memory**. [S.l.]: Morgan & Claypool Publishers, 2010. (Synthesis Lectures on Distributed Computing Theory). (Cited on page 34.)

GUERRAOUI, R.; KAPALKA, M.; VITEK, J. STMBench7: A Benchmark for Software Transactional Memory. In: ND ACM SIGOPS/EUROSYS EUROPEAN CONFERENCE ON COMPUTER SYSTEMS 2007, 2., 2007, New York, NY, USA. **Proceedings...** ACM, 2007. p.315–324. (EuroSys '07). (Cited on page 70.)

GUERRAOUI, R.; KAPAŁKA, M. Principles of Transactional Memory. **Synthesis Lectures on Distributed Computing Theory**, [S.l.], v.1, n.1, p.1–193, 2010. (Cited on page 35.)

HANSEN, P. B.; DIJKSTRA, E. W.; HOARE, C. A. R. **The Origins of Concurrent Programming**: From Semaphores to Remote Procedure Calls. Berlin: Springer-Verlag, 2002. (Cited on page 17.)

HARRIS, T.; LARUS, J.; RAJWAR, R. **Transactional Memory, 2Nd Edition**. 2nd.ed. [S.l.]: Morgan and Claypool Publishers, 2010. (Cited on pages 19, 32, 33, 34, and 35.)

HARRIS, T.; MARLOW, S.; PEYTON-JONES, S.; HERLIHY, M. Composable Memory Transactions. In: TENTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2005, New York, NY, USA. **Proceedings...** ACM, 2005. p.48–60. (PPoPP '05). (Cited on page 19.)

HARRIS, T.; MARLOW, S.; PEYTON-JONES, S.; HERLIHY, M. Composable Memory Transactions. In: TENTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2005, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2005. p.48–60. (PPoPP '05). (Cited on page 40.)

HEUVELINE, V. et al. Software Transactional Memory, OpenMP and Pthread Implementations of the Conjugate Gradients Method – A Preliminary Evaluation. In: HIGH PERFORMANCE COMPUTING FOR COMPUTATIONAL SCIENCE - VECPAR 2012, 2013, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2013. p.300–313. (Cited on pages 63 and 74.)

HONORIO, B. C. Melhorando o desempenho de aplicações transacionais através de anotações do programador. In: SPRINGER BERLIN HEIDELBERG, 2018. **Anais...** [S.l.: s.n.], 2018. (Cited on pages 23 and 54.)

Intel. **Draft Specification of Transactional Language Constructs for C++**. Disponível em: <https://sites.google.com/site/tmforcplusplus/>. (Cited on pages 54 and 55.)

INTEL. **Intel Cilk Plus**. Disponível em: <https://www.cilkplus.org/>. Acesso em: 2018-06-19. (Cited on page 59.)

KANG, S.; BADER, D. A. An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 14., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.15–24. (PPoPP '09). (Cited on page 70.)

KESTOR, G. et al. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In: PROCEEDINGS OF THE 4TH WORKSHOP ON TRANSACTIONAL COMPUTING, SER. TRANSACT '09, 2009. **Anais...** [S.l.: s.n.], 2009. (Cited on page 70.)

KESTOR, G. et al. RMS-TM: a comprehensive benchmark suite for transactional memory systems. **SIGSOFT Softw. Eng. Notes**, New York, p.335–346, 2011. (Cited on pages 19 and 52.)

LANESE, I.; MEZZINA, C. A.; SCHMITT, A.; STEFANI, J.-B. Controlling Reversibility in Higher-Order Pi. In: CONCUR 2011 – CONCURRENCY THEORY, 2011, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2011. p.297–311. (Cited on page 61.)

LARUS, J. R.; RAJWAR, R. **Transactional Memory**. [S.l.]: Morgan & Claypool Publishers, 2006. (Synthesis Lectures on Computer Architecture). (Cited on pages 32, 33, and 34.)

LUPEI, D. et al. Transactional Memory Support for Scalable and Transparent Parallelization of Multiplayer Games. In: EUROPEAN CONFERENCE ON COMPUTER SYSTEMS, 5., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.41–54. (EuroSys '10). (Cited on page 70.)

MEDIĆ, D.; MEZZINA, C. A.; PHILLIPS, I.; YOSHIDA, N. Towards a Formal Account for Software Transactional Memory. In: REVERSIBLE COMPUTATION, 2020, Cham. **Anais...** Springer International Publishing, 2020. p.255–263. (Cited on pages 61 and 93.)

MICULAN, M.; PERESSOTTI, M. **Software Transactional Memory with Interactions**. (Cited on pages 60 and 61.)

MILOVANOVIĆ, M. et al. Multithreaded Software Transactional Memory and OpenMP. In: WORKSHOP ON MEMORY PERFORMANCE: DEALING WITH APPLICATIONS, SYSTEMS AND ARCHITECTURE, 2007., 2007, New York, NY, USA. **Proceedings...** ACM, 2007. p.81–88. (MEDEA '07). (Cited on pages 64 and 74.)

MILOVANOVIĆ, M. et al. Nebelung: Execution Environment for Transactional OpenMP. **International Journal of Parallel Programming**, [S.l.], v.36, n.3, p.326–346, Jun 2008. (Cited on pages 20, 64, 71, 72, 74, 75, 77, 81, 82, 83, and 93.)

MINH, C. C.; CHUNG, J.; KOZYRAKIS, C. E.; OLUKOTUN, K. STAMP: Stanford Transactional Applications for Multi-Processing. **2008 IEEE International Symposium on Workload Characterization**, [S.l.], p.35–46, 2008. (Cited on pages 70 and 101.)

NANZ, S.; WEST, S.; SILVEIRA, K. Soares da; MEYER, B. Benchmarking Usability and Performance of Multicore Languages. In: PROC OF THE 7TH IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, 2013 ACM, 2013. **Anais...** [S.l.: s.n.], 2013. p.183 – 192. (Cited on page 73.)

NICHOLS, B.; BUTTLAR, D.; FARRELL, J. **PThreads Programming**: A POSIX Standard for Better Multiprocessing. [S.l.]: O'Reilly Media, Incorporated, 1996. (A POSIX standard for better multiprocessing). (Cited on page 28.)

OpenCilk. **Cilk multithreaded programming technology**. Disponível em: <https://cilk.mit.edu/>. (Cited on page 59.)

OPENMP. **The OpenMP API specification for parallel programming**. Disponível em: <https://www.openmp.org/>. Acesso em: 2018-07-19. (Cited on page 59.)

OpenMP Architecture Review Board. **OpenMP Application Program Interface**. [S.l.: s.n.], 2011. Specification. (Cited on page 40.)

OWENS, S. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In: ECOOP 2010 – OBJECT-ORIENTED PROGRAMMING, 2010, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2010. p.478–503. (Cited on page 18.)

PANKRATIUS, V.; ADL-TABATABAI, A.-R. A Study of Transactional Memory vs. Locks in Practice. In: TWENTY-THIRD ANNUAL ACM SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES, 2011, New York, NY, USA. **Proceedings...** ACM, 2011. p.43–52. (SPAA '11). (Cited on pages 37, 39, and 70.)

PANKRATIUS, V.; ADL-TABATABAI, A.-R. Software Engineering with Transactional Memory Versus Locks in Practice. **Theor. Comp. Sys.**, Berlin, Heidelberg, v.55, n.3, p.555–590, Oct. 2014. (Cited on pages 30, 31, 38, 39, and 57.)

PANKRATIUS, V.; ADL-TABATABAI, A.-R.; OTTO, F. Does Transactional Memory Keep Its Promises? Results from an Empirical Study. In: ACM, 2009. **Anais...** [S.l.: s.n.], 2009. (Cited on page 57.)

PAUDEL, J.; AMARAL, J. N. Using the Cowichan Problems to Investigate the Programmability of X10 Programming System. In: PROC OF THE 2011 ACM SIGPLAN X10 WORKSHOP, 2011, New York. **Anais...** ACM, 2011. (X10 '11). (Cited on page 73.)

REINDERS, J. **Intel Threading Building Blocks**. 1.ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007. (Cited on page 59.)

RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. Memórias transacionais: Uma nova alternativa para programaçao concorrente. **Laboratório de Sistemas de Computação–Instituto de Computação–Unicamp**, [S.l.], 2007. (Cited on pages 18, 31, and 37.)

ROSSBACH, C. J.; HOFMANN, O. S.; WITCHEL, E. Is Transactional Programming Actually Easier? **SIGPLAN Not.**, New York, NY, USA, v.45, n.5, p.47–56, Jan. 2010. (Cited on pages 37 and 70.)

ROSSBACH, C. J.; HOFMANN, O. S.; WITCHEL, E. Is Transactional Programming Actually Easier? **SIGPLAN Not.**, New York, NY, USA, v.45, n.5, p.47–56, Jan. 2010. (Cited on page 39.)

SCOTT, M. L.; SPEAR, M. F.; DALESSANDRO, L.; MARATHE, V. J. Delaunay Triangulation with Transactions and Barriers. In: IISWC, 2007. **Anais...** IEEE Computer Society, 2007. p.107–113. (Cited on page 70.)

SEBESTA, R. **Conceitos de Linguagens de Programação - 9.ed.** [S.l.]: Grupo A - Bookman, 2009. (Cited on pages 27, 28, and 29.)

SHRI, V. M. D.; RESHMA, K. The Transactional Memory. **International Journal of Scientific Research in Computer Science, Engineering and Information Technology**, [S.l.], p.13–20, feb 2019. (Cited on page 60.)

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. 8th.ed. [S.l.]: Wiley Publishing, 2008. (Cited on page 59.)

SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. **Fundamentos de sistemas operacionais**. [S.l.]: LTC, 2010. (Cited on pages 26 and 27.)

SKILLICORN, D. B.; TALIA, D. Models and Languages for Parallel Computation. **ACM Comput. Surv.**, New York, NY, USA, v.30, n.2, p.123–169, June 1998. (Cited on page 17.)

SRINATH, K. Python–The Fastest Growing Programming Language. **International Research Journal of Engineering and Technology (IRJET)**, [S.l.], v.4, n.12, p.354–357, 2017. (Cited on page 63.)

SRIVATSA, S. K.; KUMAR, C. R. RECONFIGURABLE FRAME WORK FOR CHIPMULTIPROCESSORS AND ITS APPLICATION IN MULTITHREADED ENVIRONMENT. **International Jr on Information Sciences and Computing**, [S.l.], 2012. (Cited on page 20.)

STROUSTRUP, B. **The C++ Programming Language**. 4th.ed. [S.l.]: Addison-Wesley Professional, 2013. (Cited on page 60.)

SÜSS, M.; LEOPOLD, C. Common Mistakes in OpenMP and How to Avoid Them. In: OPENMP SHARED MEMORY PARALLEL PROGRAMMING, 2008, Berlin. **Anais...** Springer, 2008. (Cited on page 49.)

SUTTER, H. The Pillars of Concurrency. **Dr. Dobbs**, [S.l.], 2007. (Cited on page 71.)

SWALENS, J.; DE KOSTER, J.; DE MEUTER, W. Transactional Actors: Communication in Transactions. In: ACM SIGPLAN INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR PARALLEL SYSTEMS, 4., 2017, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2017. p.31–41. (SEPS 2017). (Cited on page 61.)

SWALENS, J.; DE KOSTER, J.; DE MEUTER, W. Chocola: Integrating Futures, Actors, and Transactions. In: ACM SIGPLAN INTERNATIONAL WORKSHOP ON PROGRAMMING BASED ON ACTORS, AGENTS, AND DECENTRALIZED CONTROL, 8., 2018, New York, NY, USA. **Proceedings. . .** Association for Computing Machinery, 2018. p.33–43. (AGERE 2018). (Cited on page 61.)

TABASSUM; MEENU. Transactional Memory: A Review. In: ASSOCIATION FOR COMPUTING MACHINERY, 2020. **Anais. . .** [S.l.: s.n.], 2020. p.370–375. (Cited on page 62.)

TANENBAUM, A. **Sistemas operacionais modernos**. [S.l.]: Prentice-Hall do Brasil, 2010. (Cited on page 27.)

VOLOS, H. et al. XCalls: Safe I/O in Memory Transactions. In: ACM EUROPEAN CONFERENCE ON COMPUTER SYSTEMS, 4., 2009, New York, NY, USA. **Proceedings. . .** Association for Computing Machinery, 2009. p.247–260. (EuroSys '09). (Cited on page 40.)

VOLOS, H.; TACK, A. J.; SWIFT, M. M.; LU, S. Applying Transactional Memory to Concurrency Bugs. **SIGPLAN Not.**, New York, p.211–222, 2012. (Cited on page 19.)

VOLOS, H.; TACK, A. J.; SWIFT, M. M.; LU, S. Applying Transactional Memory to Concurrency Bugs. In: SEVENTEENTH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 2012, New York, NY, USA. **Proceedings. . .** Association for Computing Machinery, 2012. p.211–222. (ASPLOS XVII). (Cited on page 40.)

VOSS, M.; ASENJO, R.; REINDERS, J. SynchronizationSynchronization: Why and How to Avoid It. In: PRO TBB: C++ PARALLEL PROGRAMMING WITH THREADING BUILDING BLOCKS, 2019, Berkeley, CA. **Anais. . .** Apress, 2019. p.137–178. (Cited on page 71.)

WAMHOFF, J.-T.; RIEGEL, T.; FETZER, C.; FELBER, P. RobuSTM: A robust software transactional memory. In: SYMP ON SELF-STABILIZING SYSTEMS, 2010. **Anais. . .** [S.l.: s.n.], 2010. p.388–404. (Cited on page 19.)

WILLIAMS, A. **C++ Concurrency in Action**: Practical Multithreading. [S.l.]: Manning, 2012. (Manning Pubs Co Series). (Cited on pages 17, 18, 27, and 28.)

WILSON, G. **The Cowichan Problems**. Disponível em: <https://software-carpentry.org/blog/2010/06/the-cowichan-problems.html>. (Cited on page 53.)

WILSON, G. V. Assessing the Usability of Parallel Programming Systems: The Cowichan Problems. In: PROGRAMMING ENVIRONMENTS FOR MASSIVELY PARALLEL DISTRIBUTED SYSTEMS, 1994, Basel. **Anais. . .** Birkhäuser Basel, 1994. p.183–193. (Cited on pages 73 and 96.)

Wilson, G. V.; Bal, H. E. Using the Cowichan problems to assess the usability of Orca. **IEEE Parallel Distributed Technology: Systems Applications**, [S.l.], v.4, n.3, p.36–44, Fall 1996. (Cited on page 73.)

WILSON, G. V.; IRVIN, R. B. **Assessing and Comparing the Usability of Parallel Programming Systems**. [S.l.]: University of Toronto. Computer Systems Research Institute, 1995. (Cited on pages 23, 52, 53, and 75.)

WONG, M. et al. A Case for Including Transactions in OpenMP. In: BEYOND LOOP LEVEL PARALLELISM IN OPENMP: ACCELERATORS, TASKING AND MORE, 2010, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2010. p.149–160. (Cited on pages 20, 67, 68, 69, 72, and 74.)

WONG, M. et al. Towards Transactional Memory for OpenMP. In: USING AND IMPROVING OPENMP FOR DEVICES, TASKS, AND MORE, 2014, Cham. **Anais...** Springer International Publishing, 2014. p.130–145. (Cited on pages 18, 20, 60, 68, 69, 70, 71, 74, 75, 77, 81, 82, 83, and 93.)

YU, Z.; ZUO, Y.; XIONG, W. Concurrency Bug Avoiding Based on Optimized Software Transactional Memory. **Scientific Programming**, [S.l.], v.2019, p.1–19, 02 2019. (Cited on page 39.)

ZARDOSHTI, P. et al. Simplifying Transactional Memory Support in C++. **ACM Trans. Archit. Code Optim.**, New York, NY, USA, v.16, n.3, July 2019. (Cited on page 62.)

ZYULKYAROV, F. et al. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. **SIGPLAN Not.**, New York, NY, USA, v.44, n.4, p.25–34, Feb. 2009. (Cited on page 70.)

Apêndices

APÊNDICE A – Dados Brutos de Desempenho

Tabela 13 – Desempenho do caso: Entrada *pequena*, com 2 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,21	0,25	0,21	0,16	0,16	0,16
norm	0,08	1,3	5,44	0,04	0,41	3,75
outer	0,34	1,68	9,29	0,55	0,93	11.47
sor	0,43	0,45	0,46	0,25	0,3	0,31
thresh	0,32	15,63	44,20	0,33	19,69	71,33
vecdiff	0,11	3,345	20,54	0,11	1,02	21,74

Tabela 14 – Desempenho do caso: Entrada *média*, com 2 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,82	0,97	0,82	0,67	0,63	0,65
norm	0,43	6,56	27,5	0,17	2,14	29,26
outer	1,63	6,83	38,48	1,88	4,6	50.31
sor	2,04	2,19	2,41	0,99	1,57	1,57
thresh	1,23	62,76	176,61	1,51	87,53	290,65
vecdiff	0,21	6,92	40,94	0,28	2,23	38,82

Tabela 15 – Desempenho do caso: Entrada *grande*, com 2 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	2,49	22,61	142,85	2,03	9,07	150,75
norm	0,79	12,5	50,02	0,76	4,37	44,54
outer	9,36	30,92	154,63	15,47	22,23	216,26
sor	10,75	11,07	10,94	6,45	6,48	6,95
thresh	8,54	253,86	709,19	4,69	360,40	1127,7
vecdiff	1,1	34,63	207,33	0,79	12,04	218,21

Tabela 16 – Desempenho do caso: Entrada *pequena*, com 4 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,2	0,24	0,2	0,18	0,18	0,18
norm	0,06	0,58	6,51	0,03	0,23	9,90
outer	0,18	0,84	10,89	0,24	0,46	21,22
sor	0,24	0,25	0,26	0,14	0,14	0,14
thresh	0,18	15,34	37,46	0,27	28,80	107,51
vecdiff	0,09	1,59	26,66	0,09	0,56	48,21

Tabela 17 – Desempenho do caso: Entrada *média*, com 4 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,8	0,94	0,8	0,68	0,66	0,67
norm	0,28	3,99	31,54	0,1	1,17	54,47
outer	0,79	3,6	43,49	1,2	2,71	91,29
sor	0,96	0,96	1	0,60	0,76	0,78
thresh	0,69	60,72	141,63	0,79	113,69	426,79
vecdiff	0,17	3,6	52,17	0,22	1,17	99,75

Tabela 18 – Desempenho do caso: Entrada *grande*, com 4 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	2,15	12.96	177,37	1,75	6,45	326.23
norm	0,58	7,68	61,64	0,19	2,66	113,48
outer	4,98	16,01	171,21	10,7	13,69	366.51
sor	8,82	8,88	8,56	3,29	453,21	1704,95
thresh	6,31	235,86	562,98	3,26	453,21	1704,95
vecdiff	0,85	19,44	248,71	0,33	6,50	490,27

Tabela 19 – Desempenho do caso: Entrada *pequena*, com 8 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,2	0,24	0,2	0,18	0,18	0,18
norm	0,06	0,32	5,98	0,02	0,12	9,71
outer	0,11	0,45	10,28	0,12	0,23	19,83
sor	0,24	0,26	0,22	0,08	0,08	0,08
thresh	0,12	15,46	32,68	0,16	33,72	105,23
vecdiff	0,09	0,76	26,11	0,02	0,31	53,75

Tabela 20 – Desempenho do caso: Entrada *média*, com 8 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,78	0,93	0,78	0,67	0,66	0,67
norm	0,27	2,06	29,86	0,06	0,6	89,79
outer	0,41	2,08	41,15	0,60	1,11	89,77
sor	0,77	0,8	0,79	0,32	0,35	0,36
thresh	0,45	62,06	129,3	0,47	137.96	433,72
vecdiff	0,19	1,6	52,26	0,12	0,6	110,74

Tabela 21 – Desempenho do caso: Entrada *grande*, com 8 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	2,19	9,06	174,89	1,49	5,39	377,05
norm	0,55	4,79	59,66	0,12	1,4	120,16
outer	2,99	8,56	164,70	3,87	5,77	368,57
sor	6,70	6,74	6,93	1,48	1,45	368,57
thresh	2,89	245,43	517,44	1,79	539,3	1,70
vecdiff	0,83	10,46	254,64	0,16	3,46	563,01

Tabela 22 – Desempenho do caso: Entrada *pequena*, com 16 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,21	0,24	0,2	0,19	0,19	0,19
norm	0,06	0,2	5,14	0,04	0,12	7,34
outer	0,12	0,28	8,88	0,08	0,15	14,64
sor	0,24	0,25	0,24	0,04	0,05	0,05
thresh	0,11	15,74	30,72	0,17	30,48	74,29
vecdiff	0,1	0,4	23,02	0,02	0,28	43,65

Tabela 23 – Desempenho do caso: Entrada *média*, com 16 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,78	0,93	0,78	0,66	0,66	0,67
norm	0,30	0,9	25,75	0,06	0,47	45,66
outer	0,28	0,95	35,35	0,35	0,61	68,00
sor	0,86	0,88	0,91	0,15	0,17	0,19
thresh	0,4	63,40	118,61	0,19	125,43	307,56
vecdiff	0,19	0,77	46,06	0,03	0,49	89,92

Tabela 24 – Desempenho do caso: Entrada *grande*, com 16 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	2,48	6,80	154,90	1,58	6,10	303,13
norm	0,59	2,07	51,7	0,11	0,92	93,79
outer	1,74	5,08	140,89	1,57	2,69	279,35
sor	4,93	5,44	5,44	0,76	0,76	1,16
thresh	1,47	253,84	476,46	0,94	495,72	1738,91
vecdiff	0,80	5,17	228,94	0,11	2,10	475,91

Tabela 25 – Desempenho do caso: Entrada *pequena*, com 32 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,22	0,25	0,21	0,24	0,26	0,23
norm	0,06	0,15	4,76	0,04	0,08	7,07
outer	0,29	0,28	8,34	0,09	0,14	14,04
sor	4,25	0,28	0,26	0,07	0,09	0,07
thresh	1,13	16,13	34,60	0,14	30,93	51,19
vecdiff	0,1	0,25	22,36	0,03	0,17	39,2

Tabela 26 – Desempenho do caso: Entrada *média*, com 32 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,8	0,94	0,81	0,74	0,73	0,74
norm	0,32	0,65	23,78	0,08	0,33	39,82
outer	0,47	0,68	33	0,39	0,54	59,38
sor	0,9	0,95	0,88	0,24	0,25	0,25
thresh	0,45	66,27	126,08	0,33	122,26	209,01
vecdiff	0,19	0,47	44,73	0,06	0,33	79,2

Tabela 27 – Desempenho do caso: Entrada *grande*, com 32 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	2,42	9,15	149,65	2,42	13,27	267,01
norm	0,61	1,28	47,54	0,15	0,63	80,79
outer	0,81	4,53	131,83	1,92	2,24	241,51
sor	4,01	4,14	4,51	0,79	0,9	1,06
thresh	1,55	265,14	502,84	0,87	490,6	1221,75
vecdiff	0,86	2,5	221,55	0,19	1,43	399,66

Tabela 28 – Desempenho do caso: Entrada *pequena*, com 64 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,25	0,28	0,25	0,23	0,26	0,23
norm	0,06	0,12	3,95	0,05	0,08	7,27
outer	0,62	0,59	7,23	0,1	0,13	14,02
sor	0,24	0,27	0,26	0,11	0,16	0,1
thresh	0,14	18,28	43,06	0,13	28,35	59
vecdiff	0,09	0,15	19,05	0,04	0,12	37,71

Tabela 29 – Desempenho do caso: Entrada *média*, com 64 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	0,86	0,99	0,86	0,83	0,82	0,79
norm	0,29	0,46	19,75	0,15	0,25	38,07
outer	1,08	0,95	28,91	0,28	0,37	57,94
sor	0,91	0,97	0,95	0,44	0,47	0,4
thresh	0,45	76,62	144,62	0,28	114,12	228,72
vecdiff	0,18	0,27	38,14	0,09	0,2	75,87

Tabela 30 – Desempenho do caso: Entrada *grande*, com 64 threads no time de execução (tempo em segundos).

	Hydra			**Tekoha**		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	2,62	27,58	129,19	3,38	61,55	263,51
norm	0,58	0,89	39,8	0,24	0,45	76,39
outer	1,11	5,56	115,81	1,12	1,35	235,16
sor	3,91	3,85	3,64	1,57	1,65	1,56
thresh	1,73	317,33	569,75	0,86	452,39	777,31
vecdiff	0,85	1,32	192,33	0,22	0,79	379,75