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

Dissertação

LTMS - Lups Transactional Memory Scheduler. Um escalonador NUMA-Aware para STM.

Michael Alexandre Costa

Pelotas, 2021

Michael Alexandre Costa

LTMS - Lups Transactional Memory Scheduler. Um escalonador NUMA-Aware para STM.

Dissertação 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 Mestre em Ciência da Computação.

Orientador: Prof. Dr. André Du Bois

Pelotas, 2021

Insira AQUI a ficha catalográfica
(solicite em http://sisbi.ufpel.edu.br/?p=reqFicha)

Dedico. . .

AGRADECIMENTOS

Agradeço. . .

Só sei que nada sei.
— SÓCRATES

RESUMO

COSTA, Michael Alexandre. **LTMS - Lups Transactional Memory Scheduler. Um escalonador NUMA-Aware para STM.**. Orientador: André Du Bois. 2021. 55 f. Dissertação (Mestrado em Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2021.

Memória transacional em Software (STM) é apresentada como alternativa à sincronização utilizando locks e monitores. A STM permite ao programador escrever códigos paralelos de forma mais simples, pois é possível substituir o uso de bloqueios por blocos atômicos. Porém, com o aumento do paralelismo existe um aumento na contenção que em STM se reflete em um maior número de conflitos. Buscando otimizar o desempenho de STM muitos estudos focam na redução do número de conflitos por meio de escalonadores. Contudo, nas arquiteturas atuais também é importante considerar onde a memória do programa está alocada e como ela é acessada. Esta dissertação propõe um escalonador NUMA-Aware para STM, intitulado Lups Transactional Memory Scheduler (LTMS), que em tempo de execução coleta dados sobre a aplicação e arquitetura utilizada para otimizar a execução de STM em arquiteturas NUMA. Para isto o LTMS é dividido em 3 etapas, a primeira fornece um mecanismo de inicialização, com criação de filas que entendam a arquitetura e heurísticas de distribuição de threads, para analisar o impacto que a distribuição de threads possui sobre a aplicação. A segunda etapa contribui com um mecanismo para coletar dados em tempo de execução, nesta etapa são coletados dados sobre as threads e suas transações, os acessos à memória e a arquitetura utilizada. A terceira etapa contribui com um sistema para migração de threads em tempo de execução, no qual entra em ação após a ocorrência de um conflito, esta etapa busca agrupar threads conflitantes minimizando conflitos futuros e reduzindo o custo de acesso à memória, para a tomada de decisão desta etapa foram desenvolvidas duas heurísticas para entender o comportamento da STM em relação ao custo de latência e intensidade de conflitos. Para realização de testes o LTMS foi implementado junto a biblioteca TinySTM e foi utilizado com conjunto de benchmarks STAMP, os experimentos foram executados utilizando as diferentes heurísticas de distribuição e migração de threads desenvolvidos e comparados com a biblioteca TinySTM 1.0.5. Os experimentos apresentaram para maioria dos benchmarks menor taxa de abort e melhor tempo de execução.

Palavras-chave: Memórias Transacionais - TM. Non-Uniform Memory Access - NUMA. Escalonador.

ABSTRACT

COSTA, Michael Alexandre. **Transaction Scheduler for NUMA Architectures**. Advisor: André Du Bois. 2021. 55 f. Dissertation (Masters in Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2021.

...

Keywords: Transactional Memory - TM. Non-Uniform Memory Access - NUMA. Scheduler.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

TM	Memórias Transacionais
STM	Memórias Transacionais em Software
NUMA	Non-Uniform Memory Access
UMA	Uniform Memory Access

SUMÁRIO

1 INTRODUÇÃO

As arquitetura paralelas estão presentes em praticamente todas plataformas computacionais modernas. Processadores com múltiplos núcleos são usados para construção de computadores domésticos e supercomputadores. O paralelismo desses processadores crescem a cada dia, pois o aumento de desempenho dos computadores atuais se baseiam no desenvolvimento de arquiteturas paralelas.

A arquitetura paralela mais simples é baseada em um único barramento de acesso à memória, assim, um ou mais processadores e módulos de memória usam o mesmo barramento para comunicação. Estas arquiteturas são chamadas de UMA (*Uniform Memory Access*) pois possuem um único valor de latência indiferente de qual processador acessa o módulo de memória. Porém, existe um problema de escalabilidade à medida que o número de núcleos aumenta já que toda comunicação passa por um único barramento.

Como alternativa a arquitetura UMA temos a arquitetura NUMA (*Non-Uniform Memory Access*), e estão se tornando dominantes em servidores (CALCIU et al., 2017), a NUMA possui vários nodos, sendo que cada nodo é composto por um ou mais processadores e módulos de memória. O acesso à memória dentro de um nó é chamado acesso local e utiliza cada nó possui seu próprio barramento com sua latência, como as aplicações possuem acesso a toda memória, esta arquitetura permite que o processador de um nodo acesse o módulo de memória pertencente a outro nodo, esse acesso é denominado acesso remoto, e a latência do acesso remoto é maior que a latência de acesso local. Com isto, a distribuição da carga de trabalho se torna importante no desempenho das aplicações.

Para os programas extraírem o máximo de desempenho das arquiteturas paralelas, o código deve explorar todo poder computacional oferecido pelas unidades de processamento, porém, a programação paralela está longe de ser uma atividade fácil. O acesso à memória compartilhada é um dos cuidados que programadores devem tomar ao desenvolver programas paralelos, para isso, as linguagens fornecem mecanismos de sincronização de threads como locks. Porém, este modelo de programação não é intuitivo e é propenso a erros como *deadlocks*.

Uma alternativa para substituir o uso de locks na programação paralela à Memória Transacional (TM), este é um mecanismo de sincronização que realiza execuções atômicas e isoladas de partes compartilhadas de código. Na programação utilizando Software Transactional Memory (STM), o acesso à memória compartilhada é realizada dentro de uma transação executada automaticamente. A programação com STM permite que o programador não se preocupe com as aquisições e liberações de locks, o desenvolvedor deve apenas delimitar as seções críticas, o que facilita sua programação. Esta dissertação concentrou os estudos no uso de STM e seus escalonadores.

Os algoritmos de STM quando utilizados junto às arquiteturas atuais apresentam um aumento no número de contenção, o que gera conflitos e aborts. Para buscar solucionar e reduzir estes conflitos, muitos trabalhos concentram se em desenvolver escalonadores que limitam o número de threads ativos ou serialização a execução de threads agrupadas em uma única fila de execução. Um dos grandes desafios é o desenvolvimento de um escalonador transacional que avalie a arquitetura utilizada para proporcionar uma melhor distribuição das threads garantindo o máximo de paralelismo que a máquina pode fornecer.

1.1 Motivação

Memória Transacional (TM) é uma alternativa promissora para a computação paralela. A TM proporciona aos programadores uma maior facilidade ao desenvolver programas paralelos, sem ter que se preocupar com a aquisição e liberação de locks, assim, evitando problemas como *deadlocks*. Infelizmente em cenários com alto paralelismo aplicações que utilizam STM sofrem com um alto número de conflitos. Buscando reduzir o impacto que um alto número de conflitos gera sobre aplicações de STM esta é uma área de pesquisa ativa. Muitos trabalhos atuais buscam minimizar este impacto propondo escalonadores transacionais, no entanto muitos destes focam na redução dos conflitos por meio da redução de threads ativas no sistema.

As arquiteturas atuais possuem hierarquias de memória complexas com diferentes latências para acessos à memória. Muitos trabalhos que buscam otimizar o desempenho de STM por meio da redução de conflitos não consideram a arquitetura utilizada em suas heurísticas, sendo que bibliotecas atuais de STM são desenvolvidas para arquiteturas UMA e não consideram a distribuição de threads de acordo com a localidade de dados para otimizar o custo de acesso à memória encontrados nas arquiteturas NUMA. Um escalonador transacional que tenha conhecimento sobre a arquitetura pode extrair um melhor desempenho utilizando as informações sobre as áreas de memória acessadas pelas transações. Portanto o escalonador proposto busca ter conhecimento sobre a arquitetura e os dados acessados pela aplicação, para relacioná-los e extrair o melhor desempenho sem reduzir o paralelismo do sistema.

1.2 Contribuição

O principal objetivo desta dissertação é prover um escalonador de STM NUMA-Aware, o escalonador desenvolvido foi intitulado Lups Transactional Memory Schedule *(LTMS)*. O LTMS é ativado no início das aplicações de STM e portanto foi dividido em três etapas, onde este contribui com diferentes mecanismos para inicialização da aplicação, coleta de dados e migração de threads conflitantes. A etapa de inicialização busca criar filas de acordo com a aplicação e arquitetura, distribuindo as threads inicialmente para o sistema. A etapa de coleta de dados, entende a arquitetura utilizada, e em tempo de execução coleta as informações sobre os dados acessados pelas threads e transações. A etapa de migração é ativada apenas na ocorrência de conflitos nas transações e busca por meio dos dados obtidos previamente redistribuir a thread conflitante de forma a otimizar sua execução no futuro. Ao contrário de trabalhos anteriores, o LTMS busca reduzir o número de conflitos das aplicações sem reduzir o número de threads ativas na aplicação. Também considera as características da arquitetura utilizada e os acessos à memória em tempo de execução para não onerar a aplicação com um custo maior de latência, para fornecer estas características esta dissertação traz as seguintes contribuições:

- Desenvolveu um mecanismo para leitura da arquitetura e criação de filas com conhecimento sobre os nodos NUMA utilizados, onde cada fila utilizará um único núcleo disponível da arquitetura;
- Foi desenvolvida duas diferentes heurísticas de distribuição inicial de threads e aplicados para melhorar o estudo e compreender o impacto da distribuição inicial de threads em aplicações paralelas;
- Desenvolveu um mecanismo que em tempo de execução coleta informações sobre as threads e armazena os endereços de memória mais utilizados por cada thread, para então mensurar com base nos nodos NUMA utilizados os diferentes custo de acesso à memória; e
- Desenvolveu um mecanismo de migração de threads entre as filas de execução, que permite a serialização de threads conflitantes e não reduz o paralelismo da aplicação; e
- Foi desenvolvida duas heurísticas de migração e adicionadas ao escalonador para estudar e compreender o impacto que a redução da latência e o alto índice de conflitos possuem sobre aplicações de STM.

1.3 Estrutura do Texto

O trabalho está organizado da seguinte forma. O Capítulo 2 apresenta o conceito de Memória Transacional e suas características, a biblioteca TinySTM e conjunto de benchmarks STAMP utilizados neste trabalho. No Capítulo 3 abordamos os trabalhos relacionados, às suas características e classificações. O Capítulo 4 explica a implementação realizada no escalonador LTMS proposto para esta dissertação. No Capítulo 5 é apresentada a metodologia e os resultados obtidos nos testes. Por fim, o Capítulo 6 apresenta as considerações finais e trabalhos futuros.

2 MEMÓRIAS TRANSACIONAIS

Memória Transacional, ou *Transactional Memory* (TM), é uma classe de mecanismos de sincronização que fornece uma execução atômica e isolada de alterações em um conjunto de dados compartilhados. Estas estão sendo desenvolvidas para que no futuro tornem-se o principal meio de fazer a sincronização em um programa concorrente, substituindo a sincronização baseada em *locks* (MORESHET; BAHAR; HERLIHY, 2006). As TMs podem ser implementadas em *software* (STM), em *hardware* (HTM) ou ainda em uma versão híbrida de *hardware* e *software*.

Na programação utilizando STMs, todo o acesso à memória compartilhada é realizado dentro de transações e todas as transações são executadas atomicamente em relação a transações concorrentes.

A principal vantagem na programação usando STM é que o programador apenas delimita as seções críticas e não é necessário preocupar-se com a aquisição e liberação de *locks*. Os *locks*, quando utilizados de forma incorreta, podem levar a problemas como *deadlocks* (BANDEIRA, 2010).

2.1 Propriedades

Transação é uma sequência finita de escritas e leituras na memória executada por uma *thread* (HERLIHY; ELIOT; MOSS, 1993), e deve satisfazer três propriedades:

- **Atomicidade**: cada transação faz uma sequência de mudanças provisórias na memória compartilhada. Quando a transação é concluída, pode ocorrer um *commit*, tornando suas mudanças visíveis a outras *threads* instantaneamente, ou pode ocorrer um *abort*, fazendo com que suas alterações sejam descartadas;

- **Consistência**: as transações devem garantir que um sistema consistente deve ser mantido consistente. Esta propriedade está relacionada com o conceito de invariância;

- **Isolamento**: as transações não interferem nas execuções de outras transações, assim parecendo que elas são executadas serialmente. Uma transação não

observa o estado intermediário de outra.

Para garantir essas propriedades, as TMs utilizam mecanismos como o de Versionamento de Dados e Detecção de Conflitos. Estes mecanismos são utilizados pelas transações para garantir a execução das TMs.

2.2 Versionamento de Dados

O versionamento de dados faz é responsável pelo gerenciamento das versões dos dados. Ele armazena tanto o valor do dado no início de uma transação como também o valor do dado modificado durante a transação, isso para garantir a propriedade de atomicidade (BALDASSIN, 2009).

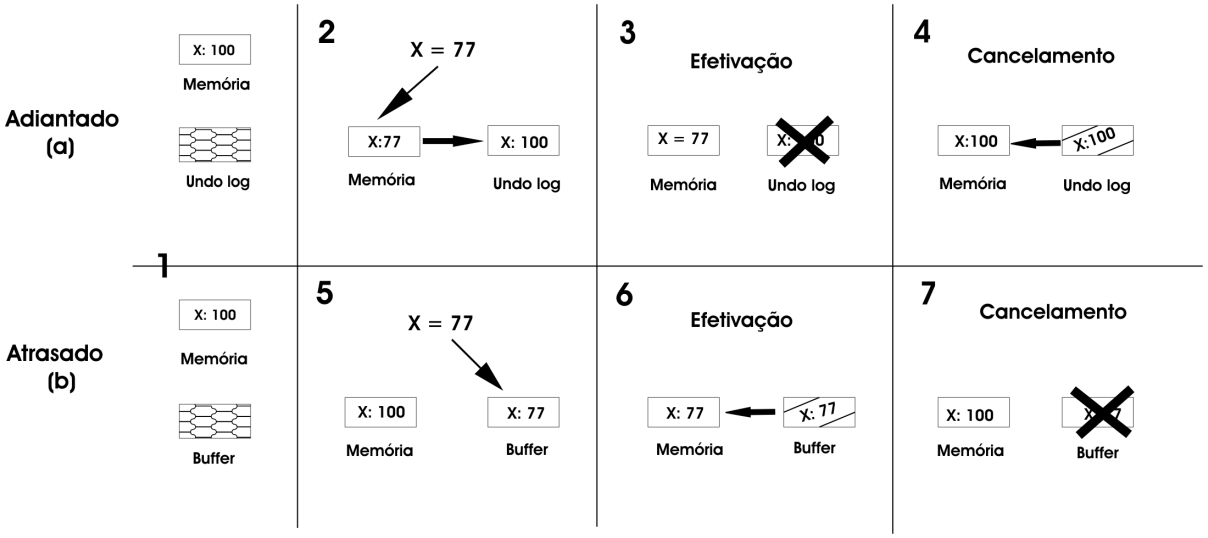


Figura 1 – Exemplo de versionamento adiantado (a) e atrasado (b). Fonte: (BALDASSIN, 2009)

Existem dois tipos de versionamento de dados:

- **Versionamento Adiantado**: como pode ser visto na Figura 1 (a), o valor modificado durante a transação é armazenado direto na memória e o valor inicial é armazenado em um *undo log*, para que no caso de cancelamento na transação o valor inicial seja restaurado na memória.
- **Versionamento Atrasado**: como pode ser visto na Figura 1 (b) neste versionamento o valor modificado durante a transação é armazenado em um *buffer* e o valor inicial é mantido na memória até que aconteça um *commit* na transação, onde o valor armazenado no *buffer* é escrito na memória. Caso aconteça o cancelamento na transação, o valor do *buffer* é descartado.

2.3 Detecção de Conflito

Mecanismos de detecção de conflitos verificam a existência de operações conflitantes durante uma transação. Um conflito ocorre quando duas transações estão acessando um mesmo dado na memória e pelo menos uma das transações está fazendo uma operação de escrita (BALDASSIN, 2009).

Da mesma forma que o versionamento de dados, a detecção de conflito também pode ser de dois tipos:

- **Detecção de Conflitos Adiantado**: ocorrem no momento em que duas transações acessam um mesmo dado e uma delas faz uma operação de escrita. Essa operação de escrita é detectada e então uma transação é abortada. Neste tipo de detecção pode ocorrer um problema chamado de *livelock*, quando duas transações ficam cancelando-se, desta forma, a execução do programa não progride. A Figura 2 mostra como é feita a detecção de conflitos adiantado.

  O Caso 1, mostra a execução sem conflitos, onde as duas transações são executadas sem problemas. Já o Caso 2, mostra o que acontece quando ocorre um conflito, onde T1 lê A e logo depois T2 escreve em A, então o conflito é detectado e T1 é abortada, após ser efetivada T2, a transação T1 consegue ler A sem problema de conflito. Por fim, o Caso 3 mostra a situação de *livelock*, onde as duas transações tentam ler e escrever em A, assim as duas acabam sempre se abortando.

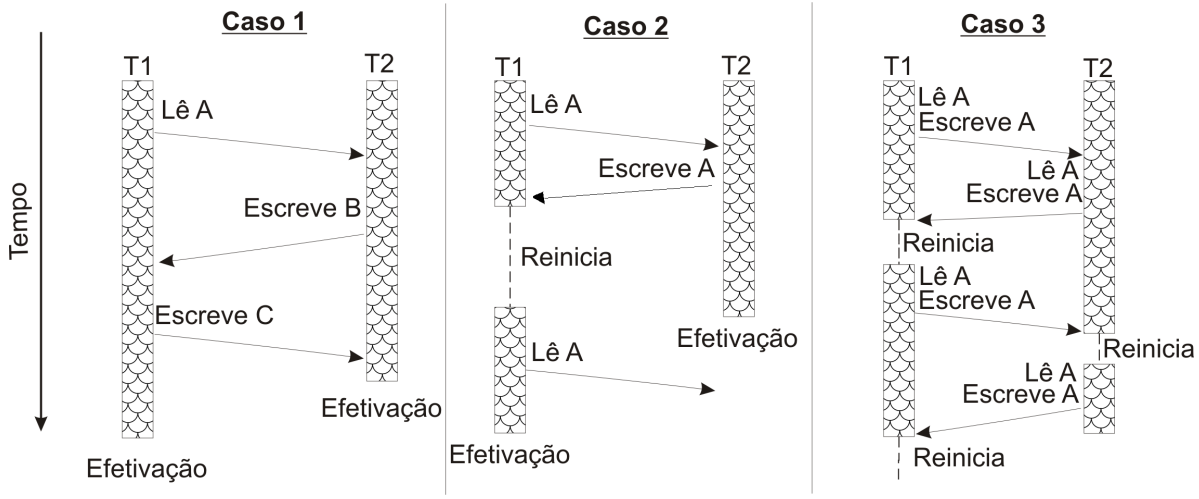


Figura 2 – Detecção de conflitos em modo adiantado. Fonte: (RIGO; CENTODUCATTE; BALDASSIN, 2007)

- **Detecção de Conflitos Atrasado**: Este tipo de detecção de conflito ocorre no final da transação. Antes da transação ser efetuada, é verificado se ocorreu um conflito. Caso tenha ocorrido, a transação é cancelada, se não é efetivada. Para transações muito grandes não é recomendado este tipo de detecção, pois uma

transação grande pode ser abortada várias vezes por transações pequenas, assim gastando tempo de processamento desnecessário, este problema se chama *starvation*. A Figura 3 mostra como é feita a detecção de conflitos atrasados.

O Caso 1, mostra as transações acessando dados diferentes, não ocasionando conflitos. No Caso 2, T2 lê A que é escrita por T1. A T2 só nota o conflito quando T1 é efetivado. Logo depois de notar o conflito, T2 é abortada. No Caso 3 não ocorre nenhum conflito, pois T1 lê A antes de T2 escrever. O Caso 4 mostra a situação em que, após ser cancelada, T1 volta a executar.

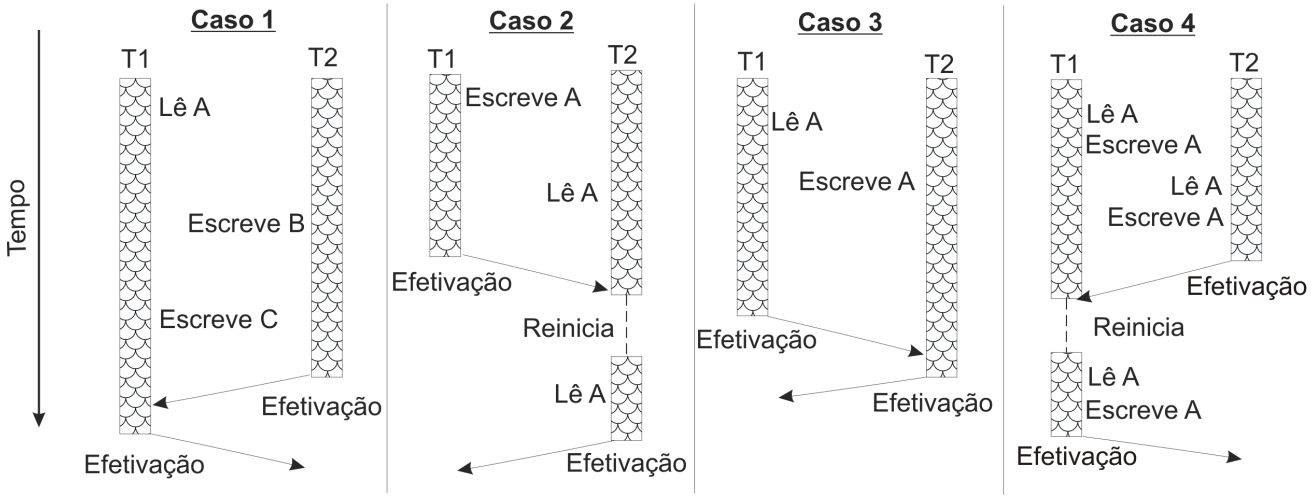


Figura 3 – Detecção de conflitos em modo atrasado. Fonte: (RIGO; CENTODUCATTE; BALDASSIN, 2007)

Para solucionar o problema de qual transação continuará executando, quando ocorre um conflito, é utilizado um gerenciador de contenção (HARRIS; LARUS; RAJWAR, 2010). O gerenciador de contenção é o responsável por decidir quando e qual transação vai ser abortada, isso para garantir que a execução do programa prossiga sem problemas.

2.4 TinySTM

A *TinySTM* (FELBER; FETZER; RIEGEL, 2008) é uma implementação de STM para as linguagens C e C++. Seu algoritmo é baseado em outros algoritmos de STM como o TL2 (*Transactional Locking* 2) (DICE; SHALEV; SHAVIT, 2006). Ela é uma biblioteca utilizada para escrever aplicativos que usam memórias transacionais para sincronização, em substituição aos tradicionais *locks*.

2.4.1 Sincronização e Versionamento

Na *TinySTM* a sincronização é feita a partir de um *array* de *locks* compartilhado que gerencia o acesso concorrente à memória. Cada *lock* é do tamanho de um ende-

reço da arquitetura (FELBER; FETZER; RIEGEL, 2008), e bloqueia vários endereços de memória. O mapeamento é feito por meio de uma função *hash*. A Figura 4 apresenta as estruturas de dados utilizadas nesta implementação.

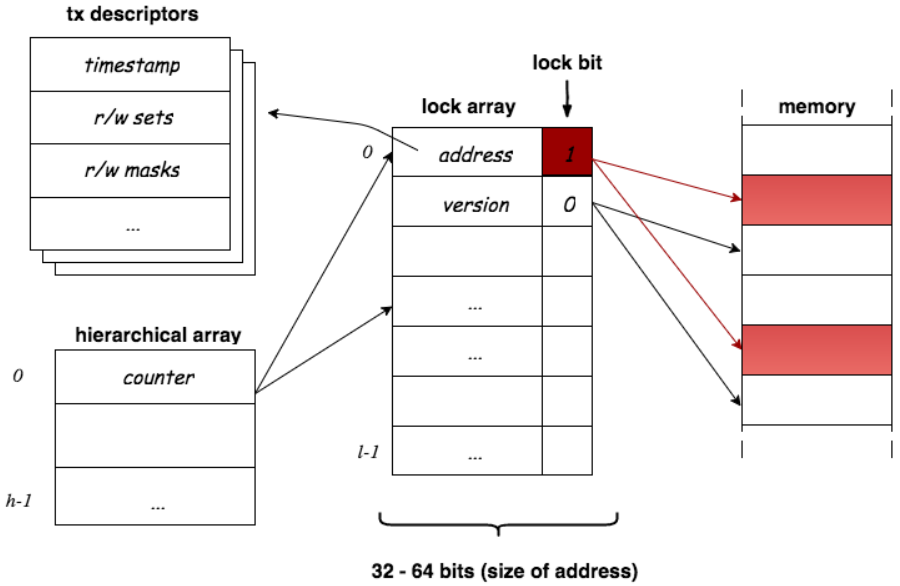


Figura 4 – Estruturas de dados utilizadas na *TinySTM*. Fonte: (FELBER; FETZER; RIEGEL, 2008)

O bit menos significativo é utilizado para indicar se o *lock* está em uso. Se o bit menos significativo indicar que o *lock* não está em uso, nos bits restantes são armazenados um número de versão que corresponde ao *timestamp* da transação que escreveu por último em um dos locais de memória abrangidos pelo *lock*.

Se o bit menos significativo indica que o *lock* está em uso, então nos bits restantes é armazenado um endereço que identifica a transação que está utilizando o dado (isso utilizando o versionamento adiantado), ou uma entrada no *write set* da transação que está utilizando o dado (isso utilizando o versionamento atrasado). Em ambos os casos os endereços apontam para uma estrutura que é *word-aligned* e seu bit menos significativo é sempre zero, por isso, o bit menos significativo pode ser utilizado como bit de bloqueio.

Quando utilizado o versionamento atrasado, o endereço armazenado no *lock* permite uma operação rápida para localizar as posições de memória atualizadas abrangidas pelo *lock*, no caso de serem acessados novamente pela mesma transação. Em contraste, a TL2 deve verificar o acesso à memória se a transação atual ainda não escreveu neste endereço, o que pode ser caro quando *write sets* são grandes. A leitura depois da escrita não é um problema quando é utilizado o versionamento adiantado porque a memória sempre contém o último valor escrito na memória pela transação ativa.

A *tinySTM* apresenta três estratégias de versionamento distintas que podem ser

utilizadas, sendo que duas utilizam versionamento atrasado (write-back) e uma utiliza versionamento adiantado (write-through), estas são:

- **Write_Back_ETL**: esta estratégia implementa o versionamento atrasado com *encounter-time locking*, isso é, o *lock* é adquirido após ocorrer uma operação de escrita e atualiza o *buffer*. O valor é escrito na memória no momento do *commit* da transação;
- **Write_Back_CTL**: esta estratégia implementa o versionamento atrasado com *commit-time locking*,isto é, ele adquire o *lock* antes de ocorrer o um *commit* e atualizar o *buffer*. Assim como no *Write-Back-ETL* o valor é escrito na memória no momento do *commit* da transação;
- **Write_Through**: esta estratégia implementa o versionamento adiantado com *encounter-time locking*, isto é, o valor é escrito direto na memória e mantém um *undo log*, caso ocorra um *abort* na transação é possível restaurar o valor anterior na memória.

A *TinySTM* utiliza *Write_Back_ETL* como sua estratégia de versionamento padrão.

2.4.2 Escritas

Quando ocorre uma escrita em um local da memória, a transação primeiro identifica o *lock* correspondente ao endereço de memória e lê o valor. Se o *lock* está em uso a transação verifica se é a proprietária do *lock* utilizando o endereço armazenado nos restantes bits de entrada. Caso a transação seja a proprietária então ela simplesmente escreve o novo valor e retorna. Caso contrário, a transação pode esperar por algum tempo ou abortar imediatamente. A *TinySTM* utiliza a última opção como padrão em sua implementação.

Se o *lock* não está em uso, a transação tenta adquiri-lo para escrever o novo valor na entrada utilizando uma operação atômica *compare-and-swap*. A falha indica que outra transação adquiriu o *lock* nesse meio tempo, então a transação é reiniciada.

2.4.3 Leituras

Quando ocorre uma leitura na memória, a transação deve verificar se o *lock* está em uso ou se o valor já foi atualizado concorrentemente por outra transação. Para esse fim, a transação lê o *lock* correspondente ao endereço de memória. Se o *lock* não tem proprietário e o valor (número de versão) não foi modificado entre duas leituras, então o valor é consistente.

2.4.4 Gerenciamento de Memória

A *TinySTM* utiliza um gerenciador de memória que possibilita qualquer código transacional utilizar memória dinâmica. As transações mantém o endereço da memória alocada ou liberada. A alocação de memória é automaticamente desfeita quando a transação é abortada, já a liberação não pode ser desfeita antes do *commit*. Contudo uma transação pode somente liberar memória depois de adquirir todos os *locks*, assim, um *free* é semanticamente equivalente a uma atualização.

2.4.5 Gerenciador de Contenção

A *TinySTM* implementa quatro estratégias de gerenciador de contenção, estas são:

- **CM_Suicide**: nesta estratégia a transação que detecta o conflito é abortada imediatamente;
- **CM_Delay**: esta estratégia assemelhasse a *CM_Suicide*, porém, espera até que a transação que gerou o *abort* tenha liberado o *lock*, então reinicia a transação. Isto porque por intuição a transação a transação que foi abortada irá tentar adquirir o mesmo *lock* novamente, provavelmente falhando em mais de uma tentativa. Está estratégia aumenta as chances de que a transação tenha sucesso sem gerar um grande número de *aborts*, melhorando o tempo de execução do processador;
- **CM_Backoff**: também parecida com a *CM_Suicide*, esta estratégia espera um tempo randômico para reiniciar a transação. Este tempo de espera é escolhido ao uniformemente ao acaso em um intervalo cujo tamanho aumenta exponencialmente a cada reinicialização;
- **CM_Modular**: esta estratégia implementa vários gerenciadores de contenção, que são alternados durante a execução. Os gerenciadores utilizados são:
  - **Suicide**: a transação que descobriu o conflito é abortada;
  - **Aggressive**: é o inverso da *Suicide*, a transação abortada é a outra e não a que descobriu o conflito;
  - **Delay**: a mesma que a *Suicide*, mas aguarda pela resolução do conflito para reiniciar a transação;
  - **Timestamp**: a transação mais nova é abortada.

A *TinySTM* utiliza a *CM_Suicide* como sua estratégia padrão de gerenciamento de contenção.

2.5 STAMP

Stanford Transactional Applications for Multi-Processing (MINH et al., 2008) é um conjunto de *benchmarks* criado para pesquisa de memórias transacionais, composto por oito *benchmarks*. Apesar de desenvolvido para a STM TL2, com algumas modificações disponíveis, pode ser usado no *TinySTM*.

O conjunto de *benchmarks* STAMP implementa vários *benchmarks*, assim, atingindo uma maior área de aplicações das STM e é o conjunto de *benchmark* mais utilizado na pesquisa de STM. Os oito benchmarks apresentados são:

- **Bayes**: Apresenta uma rede bayesiana de aprendizado;
- **Genome**: Implementa uma aplicação que reconstrói a sequência de um gene a partir de sequências maiores;
- **Intruder**: Simula o Design 5 do *Network Intrusion Detection System* (NIDS) (HAAGDORENS; VERMEIREN; GOOSSENS, 2005);
- **Kmeans**: *K-means* é um algoritmo comumente usado para partição de itens de dados em subconjuntos relacionados;
- **Labyrinth**: Implementa um algoritmo que descobre o menor caminho entre um ponto inicial e um ponto final;
- **SSCA2**: É composto por quatro *kernels* que operam em um grande, dirigido e ponderado gráfico;
- **Vacation**: Implementa um sistema de reserva de viagens alimentado por um banco de dados não-distribuído; e
- **Yada**: Implementa o algoritmo de Ruppert (RUPPERT, 1995) para refinamento de malha.

Neste trabalho vai ser utilizada a versão 0.9.10 do STAMP para avaliar e comparar a execução da biblioteca de STM TinySTM atual e a utilização do escalonador proposto.

3 ESCALONADORES

O uso de escalonadores provem melhorias nas execuções de programas, pode-se utilizar escalonadores de tarefas para melhorar o desempenho de arquiteturas, como visto no trabalho (FAVARETTO, 2014), consegue-se utilizar um escalonador para reduzir a latência de acesso à memória pelo processador em arquiteturas *NUMA*.

Em STM o uso de escalonadores pode reduzir o número de conflitos gerados pelo aumento do paralelismo, como podemos ver em (NICÁCIO; BALDASSIN; ARAÚJO, 2012) o *LUTS* apresenta heurísticas de detecção de conflitos para que o escalonador de transações evite *aborts* no decorrer de sua execução.

Escalonadores fornecem diferentes abordagens para cada problema proposto, estas distintas abordagens permitem aos desenvolvedores explorar heurísticas de escalonamento que se adaptam a arquitetura utilizada propiciando uma solução mais eficiente.

Existem muitas heurísticas diferentes para prever conflitos, estas podem servir como base para o escalonamento de transações. Para este trabalho foram estudados algumas das principais heurísticas e suas classificações.

O trabalho apresentado em (DI SANZO, 2017) fornece uma categorização dos escalonadores de STM, na qual os algoritmos são classificados de acordo com suas heurísticas.

Está categorização é dividida em algoritmos Baseados em Heurística e algoritmos Baseados em Modelo. Cada categorização possui classificações de acordo com o comportamento de sua heurística.

- Baseado em Heurística:
  - Feedback: Utiliza o feedback da execução para realimentar sua heurística;
  - Predição: Utiliza uma predição das informações para tomada de decisão;
  - Reativo: Só executa sua heurística após determinado comportamento da aplicação ocorrer; e
  - Heurística Mista: Mescla as classificações anteriores para otimizar a heurística.

Tabela 1 – Algoritmos e técnicas de escalonamento

Escalonador	Técnica
ATS	Feedback
Probe	Feedback
F2C2	Feedback
Shrink	Predição
SCA	Predição
CAR-STM	Reativo
RelSTM	Reativo
LUTS	Heurística Mista
ProVIT	Heurística Mista
SAC-STM	Aprendizado de Máquina
CSR-STM	Modelo Analítico
MCATS	Modelo Analítico
AML	Modelo Misto

- Baseado em Modelo:
  - Aprendizado de Máquina: Utiliza algoritmos de aprendizado de máquina para tomar decisão;
  - Modelo Analítico: Monta modelos analíticos para tomar decisão; e
  - Modelo Misto: Mistura as classificações acima para otimizar a heurística.

A tabela 1 apresenta as classificações dos principais algoritmos revisados na bibliografia durante o desenvolvimento deste trabalho.

3.1 ATS

Adaptive Transaction Scheduling (ATS) (YOO; LEE, 2008) foi um dos primeiros trabalhos a apresentar um escalonador de MT para trabalhar junto com gerenciador de contenção.

O ATS utiliza um valor para tomada de decisão denominado CI (Contention Intensity), cada thread em execução possui seu próprio CI. O CI é calculado cada vez que ocorre um commit ou um abort e é zerado a cada início de transação.

O escalonador utiliza o valor do CI em sua tomada de decisão. Quando o valor do CI ultrapassa um limiar pré definido, a thread é colocada em uma única fila para garantir uma execução de forma serial.

3.2 CAR-STM

Collision Avoidance and Resolution (CAR-STM) (DOLEV; HENDLER; SUISSA, 2008) foi desenvolvido para evitar que conflitos já existentes voltem a ocorrer. Para isto é apresentado duas heurísticas de gerenciamento.

A primeira heurística é denominada Básica e busca executar de forma serial as transações conflitantes sem manter um histórico da execução. A segunda, denominada Permanente, busca manter um histórico das transações que conflitaram e executá-las de forma serial.

- Básica: Quando detectado um conflito, a transação mais recente é abortada e migrada para fila da transação conflitante, assim sua execução será serializada.
- Permanente: Quando uma transação Tb aborta em relação a Ta, Tb é migrado para fila de Ta e sua ordem de execução será Ta -> Tb. Caso a transação Ta conflite e aborte em relação a Tc, Ta deverá ser migrada para fila de Tc carregando sua dependência Ta -> Tb.

3.3 LUTS

Light-Weight User-Level Transaction Scheduler (LUTS) (NICÁCIO; BALDASSIN; ARAÚJO, 2012) apresenta um escalonador que busca evitar a ociosidade de um núcleo após a serialização de uma transação.

Para isto, cada thread é representado internamente por um Registro de Contexto em Execução (RCE), cada RCE encapsula uma thread. No início da execução o escalonador cria uma fila de RCEs para serem executados no futuro.

Assim, o LUTS dispara um RCE por núcleo, e utiliza a fila para não disparar mais RCEs que núcleos disponíveis. Cada REC disparado é convertido em uma thread de sistema que executa um conjunto de transações.

Na tentativa de evitar conflitos, o LUTS apresenta uma forma dinâmica para solucioná-los, considerando transações curtas e transações longas. Para definir o tamanho da transação é utilizada a contagem de ciclos da mesma, onde a partir de 100 mil ciclos temos uma transação longa.

Para transações curtas, a heurística utilizada é similar a do ATS, o escalonador calcula a intensidade de conflito da transação e serializa esta quando o cálculo ultrapassa um limiar. Porém o LUTS escolhe outra transação para substituir a atual.

Para transações longas, a heurística é mais elaborada, utilizando três metadados globais:

- activeTx: Um vetor de tamanho igual ao total de núcleos disponíveis, usado para armazenar o identificador da transação que está sendo executada.
- conflictTable: Uma tabela do histórico de conflitos, cada linha armazena um conjunto de transações dada pelo activeTx, e cada coluna armazena a probabilidade de conflito.

- bestTx: Um vetor que sumariza a melhor transação a ser executada para cada núcleo.

Quando uma transação realiza um commit ou abort o escalonador se encarrega de atualizar a conflictTable na sua respectiva linha, aumentando ou diminuindo sua probabilidade de conflito.

Para evitar percorrer a conflictTable no início de cada transação o LUTS percorre a bestTx e seleciona qual transação deve executar. Quando a conflictTable é atualizada o escalonador atualiza a bestTx.

3.4 Shirink

Shirink (DRAGOJEVIć et al., 2009) apresenta um escalonador que busca minimizar a ocorrência de aborts com base nos conjuntos de leituras e escritas ocorridos em cada thread.

O escalonador é baseado em predição e usa como heurística os acessos à memória das transações executadas anteriormente. Para evitar overhead em sua execução o Shirink avalia os acessos apenas se existir uma alta contenção no sistema.

No início de cada transação o escalonador avalia se existe a relação entre commit e abort é superior a um limiar predefinido, se esse valor for superior ao limiar o escalonador considera que o sistema possui uma alta contenção e ativa a heurística para serializar as transações em execução.

Cada thread possui um conjunto dos acessos de leitura e escrita realizados pelas transações, quando uma transação vai iniciar com um sistema de alta contenção esse conjunto de leitura e escrita é verificado, se outra thread em execução possuir um conjunto semelhante o escalonador assume que há uma alta chance da transação abortar.

Para que a transação não aborte a thread que iniciaria a transação é bloqueada até o fim da transação na thread em execução, assim o Shrink busca forçar a serialização das execuções.

3.5 ProVit

O escalonador *ProVIT* (RITO; CACHOPO, 2015) fornece uma abordagem otimista da execução, evitando considerar que toda transação que abortou ira abortar novamente na sequência.

Assim como o LUTS o ProVIT avalia o tamanho das operações atômicas para aplicar sua heurística. Porém no ProVIT mais de uma heurística pode estar ativa ao mesmo tempo.

Também foi apresentada a observação que duas transações conflitantes, de leitura

e escrita, podem efetuar o commit dependendo da ordem, se o commit for efetuado primeiro pela transação de leitura, a de escrita não será conflitante.

Operações atômicas longas utilizam uma política baseada em grão fino para melhorar a precisão da predição e evitar a reexecução de transações. Essa predição utiliza como base o conjunto de leitura das transações já executadas.

Se uma transação efetua um abort o escalonador marca esta transação como Very Important Transaction (VIT) e copia seu conjunto de leitura para uma lista auxiliar global. Quando uma transação tenta efetuar um commit, ela verifica a lista global para garantir que não há conflito entre os conjuntos de escrita e leituras.

Caso haja conflito entre a transação e alguma VIT, o commit é adiado por um tempo pré-determinado. Assim, o escalonador tenta garantir que as VITs não abortem novamente. Caso não haja conflito o commit é realizado.

Nas operações atômicas curtas a heurística evita a validação com base na intersecção para não adicionar overhead desnecessário. Sendo assim, ela apresenta uma ideia similar a do ATS, onde é utilizada uma métrica de decisão para serializar as transações.

Para definir quando uma transação será serializada, o escalonador utiliza um valor calculado em tempo de execução denominado Tempo Perdido (TP). Cada operação atômica possui seu próprio TP, que é calculado com base em um valor pré-definido e a quantidade de reexecução da transação e seu TP anterior.

Toda operação atômica começa com TP igual a zero e é executada livremente, conforme o TP aumenta o ProVit se encarrega de serializar as transações reduzindo a concorrência até o ponto em que somente uma transação poderá ser executada por vez.

Para definir se uma operação atômica é curta ou longa foi implementada uma política de definição, onde toda operação atômica inicia sua execução como curta. Quando uma transação finaliza o escalonador atualiza seu conhecimento sobre as operações atômicas com base no TP.

Uma operação atômica é considerada longa quando o tamanho médio do seu conjunto de leitura for maior que um limite pré-definido. Assim, o tamanho da operação atômica é baseado nas operações de leitura e não no tempo de execução.

3.6 STMap

O *STMap* (PASQUALIN et al., 2020a) apresenta um escalonador que utiliza mecanismos de sharing-aware mapping para aplicações STMs. O escalonador utiliza esse mecanismo para executar as transações concorrentes no mesmo núcleo NUMA para otimizar a execução da aplicação.

Em tempo de execução o escalonador coleta dados sobre os comportamentos

compartilhados entre as threads, esses dados são usados para calcular um mapeamento otimizado de threads para núcleos e migra as threads em execução.

A coleta de dados é realizada em tempo de execução quando ocorre uma escrita ou leitura dentro de uma transação, esse endereço de leitura ou escrita é comparado com os endereços utilizados pelas outras transações. Se esse endereço é usado por outra transação o STMap incrementa uma matriz de comunicação com essas informações.

A matriz de comunicação possui como tamanho o mesmo número de threads em execução. Supondo que a transação t1 executada pela thread T1 manipule o mesmo endereço que a transação t2 da thread T2, a transação t1 descobre o ID da thread T1 e da thread T2 e incrementa o valor da matriz de comunicação nas posições respectivas aos IDs.

Essa matriz é utilizada para caracterizar as transações e mapear a arquitetura em relação às threads, assim sendo possível agrupar as threads por nodos de processamento para otimizar a execução e aproveitar ao máximo a coerência de cache.

4 LTMS - LUPS TRANSACTIONAL MEMORY SCHEDULER

Memórias transacionais fornecem um nível maior de abstração para o desenvolvimento de programas paralelos, e conforme visto no capítulo anterior, existem vários trabalhos que focam em desenvolver escalonadores que compreendem a aplicação para extrair melhor desempenho.

Porém, os escalonadores atuais não consideram as diferenças entre as arquiteturas paralelas existentes. O escalonador LTMS, proposto neste trabalho, se propõe a avaliar a aplicação e a arquitetura em tempo de execução para tirar máximo proveito do paralelismo existente.

As duas principais arquiteturas paralelas que serão abordadas na proxima seção são, arquiteturas UMA e arquiteturas NUMA. Os escalonadores atuais assim como as bibliotecas de STM são pensados para arquiteturas UMA não considerando as diferenças quando executadas em NUMA.

O LTMS foi desenhado para acompanhar toda execução de uma aplicação que utiliza STM. Sendo assim, inicialmente ele prove filas de execução e implementa heurísticas de distribuição de threads entre as filas no início da aplicação. Além disso, para entender a arquitetura são coletados os endereços de escrita e leitura, e os números de aborts e commits das threads e transações em tempo de execução, estes dados são utilizados nas heurísticas que definem quando uma thread deve ser migrada para reduzir latência de acesso a memória ou reduzir a contenção gerada na aplicação.

4.1 Motivação

Máquinas *NUMA* têm a vantagem de agregar maior paralelismo ao adicionar mais processadores sem aumentar o gargalo de acesso ao barramento. Sua arquitetura é feita para que os processadores não utilizem o mesmo barramento de acesso à memória como é feito em arquiteturas *UMA*.

As arquiteturas *NUMA* possuem múltiplos núcleos dispostos em conjuntos de processadores (Nodos), a memória é fisicamente composta por vários bancos de memória, podendo estar cada um deles vinculados a um Nodo e a um espaço de endereça-

mento compartilhado. Quando o processador acessa a memória que está vinculada a si, diz-se que houve um acesso local. Se o acesso for à memória de outro processador, diz-se que ocorreu um acesso remoto. Os acessos remotos são mais lentos que os acessos locais, uma vez que é necessário passar pela rede de interconexão para que se consiga chegar ao dado localizado na memória remota (FAVARETTO, 2014).

Os escalonadores de STM atuais buscam reduzir o número de conflitos para reduzir a quantidade de reexecução das transações. Para isto estes escalonadores implementam filas de execução e migração de threads que tornam serial a execução das transações conflitantes, como nos trabalhos apresentados em (DRAGOJEVIć et al., 2009), (NICÁCIO; BALDASSIN; ARAÚJO, 2012), e (RITO; CACHOPO, 2015).

Algoritmos NUMA-Aware avaliam as diferentes características da arquitetura que a aplicação está executando. Possuindo conhecimento das diferentes latências de acesso à memória, esses algoritmos podem extrair o máximo de recurso da máquina. Assim, alguns algoritmos NUMA-Aware avaliam os conjuntos de leitura e escrita das aplicações para decidir qual é o melhor nodo de execução para a thread, ou quando deve ser migrado a página de memória. Outros utilizam uma matriz de comunicação para fazer um mapeamento de threads e assim otimizar sua execução, como é feito em (PASQUALIN et al., 2020b).

Os escalonadores de STM atuais não consideram a arquitetura e seu custo de acesso à memória para serializar as execuções. Alguns escalonadores de STM avaliam os conjuntos de leitura e escrita apenas com interesse em reduzir o número de conflitos, como é visto em (DRAGOJEVIć et al., 2009).

O LTMS, diferente de outros trabalhos, é um escalonador que avalia as características da arquitetura, e em tempo de execução monta uma matriz de comunicação com base nas leituras e escritas realizadas pelas threads. Está matriz de comunicação é utilizada para avaliar o custo de acesso à memória e migrar as threads em execução entre as filas, buscando diminuir os números de conflitos por meio da serialização das transações e otimizar a execução aproveitando a melhor distribuição das tarefas na arquitetura NUMA, reduzindo assim a latência de acesso à memória.

4.2 Escalonador

O LTMS é um escalonador de STM NUMA-Aware que identifica as características da arquitetura e do programa em tempo de execução para extrair o máximo de desempenho da máquina utilizada. O escalonador opera em três estágios, sendo eles, a inicialização do sistema, a coleta de dados em tempo de execução e a migração de threads.

- Inicialização do sistema: Inicialmente associa filas de execução aos processadores e implementa duas técnicas de distribuição inicial de threads;

- Coleta de dados em tempo de execução: Em tempo de execução, são coletadas informações de acesso a memória e a quantidade de commits e aborts feitas pelas transações; e
- Migração de Threads: Quando transações abortam, utiliza heurísticas baseadas nos dados coletados, para decidir se as threads devem ser migradas para outras filas.

4.2.1 Inicialização do sistema

Como podemos ver na figura 5 o escalonador LTMS é inicializado junto com a aplicação, o escalonador é responsável por ler as característica da arquitetura e criar filas de execução com base nas threads da aplicação e no número de cores disponíveis. O LTMS fornece uma biblioteca de threads integrada a STM que provê todos os recursos necessários para o desenvolvimento das aplicações, quando uma thread é criada ela fica disponível para o escalonador distribuir ela entre as filas com base em uma na Heurística de distribuição.

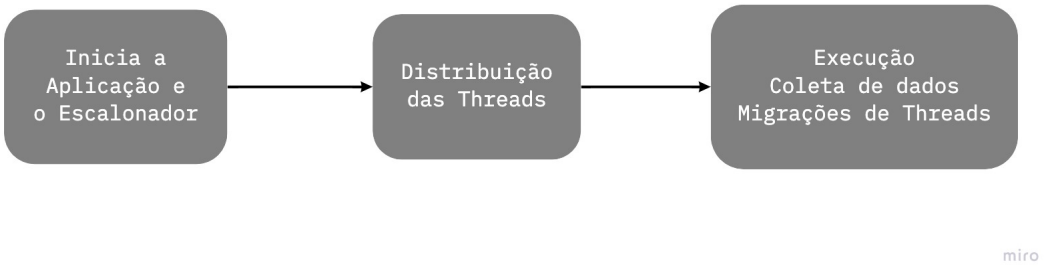


Figura 5 – Fluxo de execução da LTMS

Ao inicializar uma aplicação o número de threads utilizados vai ser passado para o escalonador na chamada de sua biblioteca de threads, assim, o LTMS compara o número de threads da aplicação com a quantidade de cores da máquina, se o número de threads da aplicação for maior que o número de cores o LTMS cria uma fila para cada core, como visto na figura 6.

Se a quantidade de threads da aplicação for menor que o número de cores disponível na arquitetura, o LTMS cria a mesma quantidade de filas que a quantidade de threads, fixando um core por fila e distribuindo uma thread por fila, como visto na figura 7

Após a criação das filas de execução, conforme apresentado acima, as threads criadas na aplicação são distribuídas com base em uma heurística de distribuição, o escalonador foi desenhado para que diferentes heurísticas possam ser desenvolvidas

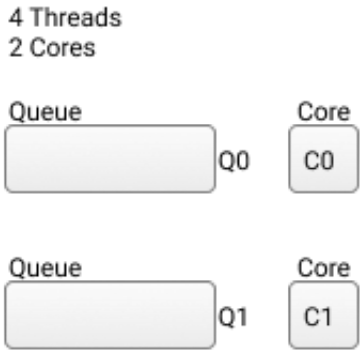


Figura 6 – Criação das filas de execução com base nos cores

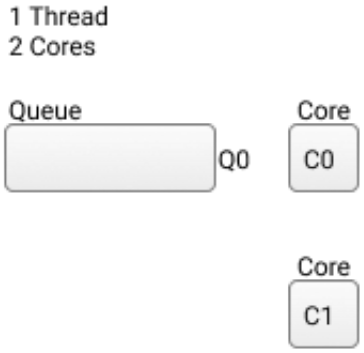


Figura 7 – Criação das filas de execução com base nas threads

e acopladas a ele, permitindo testar diferentes formas de distribuição de threads. Para este trabalho foram implementadas duas heurísticas de distribuição.

A primeira heurística implementada distribui uma thread por fila até a conclusão de todas threads disponíveis. A figura 8 traz como exemplo 4 threads e 2 cores, neste caso serão criadas uma fila para cada core.

O escalonador executará a primeira fase de distribuição, colocando uma thread para cada fila existente. Após a primeira fase o escalonador verifica se ainda possui threads a serem distribuídas, caso haja threads o LTMS repete a distribuição em uma segunda fase, até acabarem as threads.

Neste cenário a Fila intitulada Q0 fica com as threads t0 e t2, e a fila Q1 fica com as threads t1 e t3. Veja que o LTMS alocou a thread t0 em Q0 e depois alocou t1 em Q1, então voltou a execução para alocar t2 em Q0 e t3 em Q1.

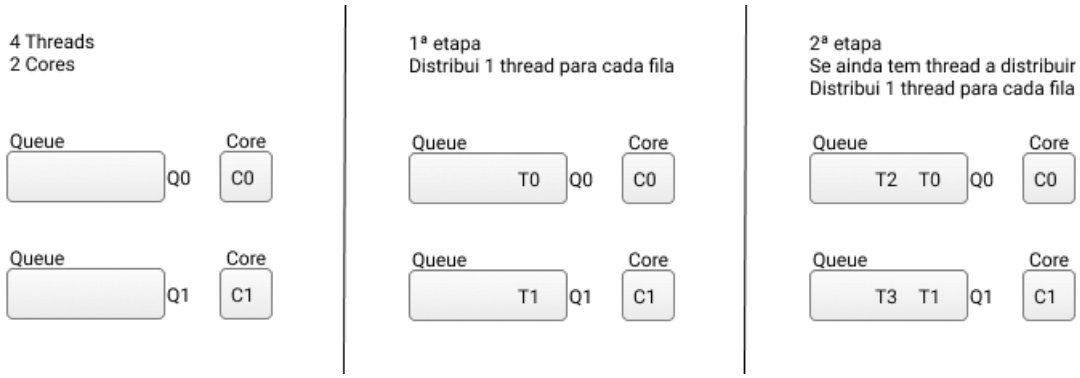


Figura 8 – Heurística um de distribuição de threads

A segunda heurística distribui chunks de threads por fila, sendo o tamanho do chunk determinado pela razão entre a quantidade de threads e a quantidade de filas. No exemplo apresentado na figura 9 temos o mesmo cenário de filas, cores e threads apresentados anteriormente.

To do (**??**)

4.2.2 Coleta de dados em tempo de execução

Durante a execução da aplicação o escalonador se encarrega de coletar dados de sua execução e da arquitetura utilizada para otimizar a redistribuição de threads entre as filas existentes caso ocorram aborts nas transações. Entre os dados coletados estão os acessos à memória, a quantidade de aborts e a quantidade de commits realizados pelos threads, também são coletadas informações sobre os nodos NUMA existentes na arquitetura e os custos de latência existentes.

Os dados coletados sobre os acessos à memória fornecem insumos para duas matrizes, uma matriz de comunicação e uma matriz de endereços. Estas matrizes serão utilizadas nas heurísticas de migração para definir o grau de relação entre as

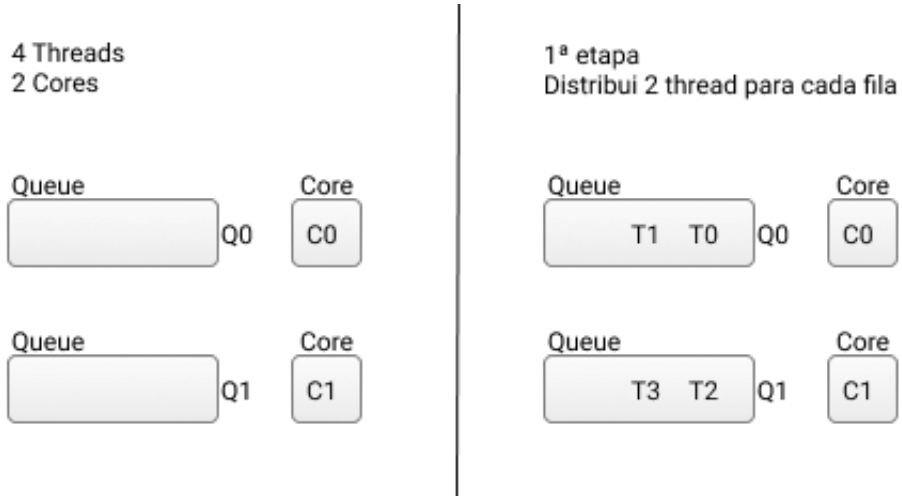


Figura 9 – Heurística dois de distribuição de threads

filas de execução para reduzir os conflitos após uma migração de threads entre as filas.

A matriz de comunicação fornece insumos sobre a quantidade de vezes que duas threads acessam o mesmo endereçamento de memória. Os valores da matriz de comunicação se dão pelos somatórios dos acessos ao mesmo endereçamento de memória entre dois threads, sendo os identificadores dos threads as posições da matriz. Sendo assim se a thread com id 0 lê ou escreve 1000 vezes no mesmo endereço que a thread de id 1, o valor da matriz de comunicação na posição *matrixComm[0][1]* será 1000.

Para evitar overhead a coleta de dados da matriz de comunicação ocorre por amostragem, neste trabalho utilizamos 1 a cada 100 acessos por thread para acionar o contador, esta mesma abordagem foi apresentada nos trabalhos (PASQUALIN et al., 2020a) e (PASQUALIN et al., 2020b). O LTMS permite que o valor da amostragem seja configurado, caso o desenvolvedor queira que todos os dados de acesso à memória sejam considerados.

A matriz de endereços por sua vez armazena a informação de quais endereços iguais foram acessados mais vezes pelas threads. Os valores da matriz de endereços são os endereços de memória acessados mais vezes entre dois threads, e assim como na matriz de comunicação os identificadores dos threads são as posições da matriz. Assim, se a thread com id 0 e a thread de id 1 possuem mais acesse em comum com o endereço 0xbff6f36c, o valor da matriz de endereços na posição *matrixAddress[0][1]* será 0xbff6f36c.

Para montar a matriz de endereços foi utilizado o mesmo sistema de amostragem apresentado anteriormente, no momento do armazenamento o endereço avaliado é armazenado em uma fila que é ordenada pela quantidade de acessos que o endereço recebeu, o endereço com maior número de acessos é armazenado na matriz de

comunicação.

O LTMS também fica encarregado de coletar dados sobre o comportamento das transações. Durante sua execução uma thread pode ter n transações que geram aborts e commits, o escalonador mantém em cada thread um contador para os commits e um para os aborts. Esse contador mantém o histórico de commits e aborts de uma thread que será utilizado para identificar o índice de contenção da thread.

4.2.3 Migração de Threads

O LTMS fornece um sistema de migração de threads entre as filas existentes, esse sistema entra em ação após a ocorrência de um abort e busca agrupar as threads conflitantes com intuito de serializadas para evitar conflitos futuros. O sistema de migração é dividido em duas etapas, a identificação da fila para a qual o thread será migrado e a heurística de migração que define se a migração irá acontecer.

A figura **??**migration) ilustra a função de migração denominada *(*migrateThread), esta função é executada após a ocorrência de um abort, a função *findBestQueue* identifica a fila para qual podemos efetuar a migração e a função *okToMigrate* utiliza uma heurística que determinar se devemos migrar a thread atual. Caso a thread deva ser migrada, a thread é adicionada à fila de execução para qual desejasse migra-la. Caso a thread não deva ser migrada, a função retorna para operação de abort e segue a execução utilizando o gerenciador de contenção.

```
migrateThread(thread) {
   if (okToMigrate(thread))
     findBestQueue(thread).push();
}
```

Figura 10 – Função de migração

A etapa de identificação das threads conflitantes busca entender a aplicação e a arquitetura para definir para qual fila a thread que gerou o abort deve ser migrada, para isso a função *findBestQueue* recebe o identificador da thread atual e consulta na matriz de comunicação qual a thread em execução possui mais acessos em comum à memória, após identificar a thread a função retorna a fila na qual a thread pertence. Após identificar a fila com maior afinidade, o LTMS utiliza uma das heurísticas para decidir se deve migrar a thread ou não.

A etapa de heurística de migração avalia se a thread que gerou o abort está apta a ser migrada, o LTMS permite que diferentes heurísticas de migração sejam desenvolvidas e acopladas a ele. Para este trabalho foram desenvolvidas duas heurísticas de migração, que avaliam os dados coletados para tomar a decisão de migrar a thread. Estas heurísticas foram denominadas *threshold* e *latency* e são apresentadas nas figuras 11 e 12.

A primeira heurística, denominada threshold, avalia o nível de contenção apresentado pela thread em tempo de execução, esse nível de contenção é medido pela razão entre os aborts e commits realizados pela thread, onde um resultado alto indica uma maior contenção ocasionada pelos aborts. Para realizar uma migração utilizando esta heurística o LTMS executa a função apresentada na figura11.

A função *thresholdHeuristic* apresentadas na figura 11 calcula o índice de contenção dado pela razão dos aborts e commits realizados pela thread e avalia se o índice de contenção é maior que um valor limiar. Se o índice de contenção for maior que o limiar a função permite a migração da thread, se o valor do índice de contenção ficar abaixo do limiar a thread não deve ser migrada.

O limiar denominado *threshold* é uma constante definida pelo desenvolvedor que indica o nível máximo de contenção aceito pela aplicação, um valor baixo para o limiar gera mais migrações que proporciona maior serialização do sistema reduzindo assim os aborts e aumentando tempo de execução, enquanto um limiar muito alto mantém o paralelismo mas aumenta o número de aborts.

```
bool thresholdHeuristic(thread) {
  return thread.aborts/thread.commits >= threshold
}
```

Figura 11 – Heurística de migração threshold

A segunda heurística de migração, denominada *latency*, avalia a latência de acesso a memória entres os nodos da filas envolvidas na migração e o endereço de memória mais acessado pela thread. Para realizar uma migração utilizando esta heurística o LTMS executa a função denominada *latencyHeuristic* apresentada na figura12.

A função *latencyHeuristic* busca na matriz de endereços qual o endereço de memória em comum é mais acessado pelas filas. A função também consulta quais os nodos NUMA a fila atual e a fila que está sendo avaliada pertencem. Com as informações sobre os nodos NUMA e o endereço de memória mais acessado, é avaliada as latências de acesso das duas filas para o endereço de memória, se a fila atual possui uma latência de acesso maior que a fila para a qual pretendemos migrar a thread o escalonador efetua a migração, caso a latência seja menor ou igual a thread mantém sua execução na fila atual.

Migrando a thread para uma fila com latência menor que a atual, o LTMS busca reduzir o número de aborts serializando parte da execução, e busca também aproveitar as características da arquitetura otimizando o acesso à memória dentro da região NUMA. A migração não ocorre se a latência da nova fila for maior para evitar futuros acessos entre diferentes nodos NUMA.

```
bool latencyHeuristic(currentQueueId, nextQueueId) {
  address = getAddress(currentQueueId, nextQueueId)
  nodeNextQueue = getNodeNuma(nextQueueId.node)
  nodeCurrentQueue = getNodeNuma(currentQueueId.node)
  currentLatency = latency(nodeCurrentQueue, address)
  nextLatency = latency(nodeNextQueue, address)
  return currentLatency > nextLatency
}
```

Figura 12 – Heurística de migração latency

4.3 Conclusão

Como visto na seção anterior o LTMS é um escalonador NUMA-Aware de tres etapas, a primeira se encarrega de inicializar a aplicação criando filas de execução e distribuindo as threads entre estas filas, a segunda etapa coleta dados sobre a arquitetura e a aplicação em tempo de execução para otimizar a aplicação por meio da serialização de threads conflitantes, a terceira etapa se encarrega de avaliar se uma thread que abortou deve ser serializada, a serialização ocorre por meio da migração da thread que realizou o abort para uma fila que possua as mesmas características de acesso à memória.

O escalonador permite a criação de diferentes heurísticas para avaliar seu comportamento de distribuição de threads na fase inicial e na migração de threads na ocorrência de aborts. Estas heurísticas podem ser criadas e acopladas ao LTMS para melhorar o fluxo de execução das aplicações e facilitar estudos futuros.

O LTMS é um escalonador reativo que realiza a fase de migração de threads a partir da ocorrência de um abort, como podemos ter mais de uma transação por thread os dados coletados para o sistema de migração são avaliados por threads, isto permite a comparação entre as características dos acessos à memória e o índice de contenção gerado por cada thread. Assim como o LTMS o Shirink (DRAGOJEVIć et al., 2009) avalia as informações em tempo de execução com base nas threads, porém este não avalia as características de acesso à memória.

Algoritmos como LUTS (NICÁCIO; BALDASSIN; ARAÚJO, 2012) utilizam a migração para reduzir o índice de contenção, porém este algoritmo realiza a migração para uma única fila, serializando a aplicação sem considerar as características de acesso à memória. Outros algoritmos, como o ATS (YOO; LEE, 2008) e o Shirink (DRAGOJEVIć et al., 2009), realizam a serialização sem efetuar uma migração, apenas controlando o número de threads ativos na aplicação o que reduz a contenção mas utiliza totalmente o recurso disponível na arquitetura.

A tabela 2 apresenta uma comparação entre as principais características do LTMS e os escalonadores apresentados neste trabalho, a tabela busca entender as diferen-

Tabela 2 – Comparativo entre os escalonadores apresentados

Escalonadores	LTMS	ATS	Shrink	LUTS	ProVIT	STMap
Distribuição inicial de threads	Sim	Não	Não	Não	Não	Sim
Coleta de dados por threads	Sim	Não	Sim	Sim	Sim	Sim
Migração entre filas	Sim	Não	Não	Sim	Não	Não
Avalia a arquitetura	Sim	Não	Não	Não	Não	Sim
Técnica de escalonamento	Reativo	Feedback	Predição	Mista	Mista	Predição

ças que o escalonador fornece para aplicações paralelas que utilizam STM.

5 EXPERIMENTOS

Para este trabalho foi desenvolvido um escalonador de STM NUMA-Aware, intitulado LTMS, que funciona em três etapas. Para validação, o escalonador LTMS foi desenvolvido em linguagem C, e aplicado à biblioteca de STM TinySTM em sua versão 1.0.5, e foram rodados experimentos com o conjunto de benchmarks STMAP em sua versão 0.9.10.

Os testes foram rodados em uma máquina de arquitetura NUMA com processador Intel Xeon E5-4650 com 96 núcleos e 192 threads em *hyper threading* e 468 Gb de memória RAM, utilizando com sistema operacional Linux Debian kernel 4.19.0-8-amd64 e gcc 8.3.0.

Como comentado anteriormente o LTMS permite que heurísticas para distribuição de threads e heurísticas para migração de threads sejam desenvolvidas e acopladas a ele de maneira simples para que estudos possam ser realizados, neste trabalho foi desenvolvido duas heurísticas de distribuição e duas heurísticas de migração. Para obter os resultados do LTMS foram rodadas quatro baterias de testes com as seguintes configurações, LTMS com distribuição Sequential e migração Threshold, LTMS com distribuição Sequential e migração Latency, LTMS com distribuição Chunks e migração Threshold, e LTMS com distribuição Chunks e migração Latency. Para comparar o desempenho do escalonador LTMS, foi executada uma bateria de teste com a TinySTM 1.0.5 sem modificações.

Cada bateria de teste consiste em 30 execuções de cada benchmark do conjunto STAMP, para os cenários de 1, 2, 4, 8, 16, 32, 64, 128, 256, e 512 threads. A seção 5.1 mostra os resultados obtidos com os experimentos.

5.1 Resultados

Para validação deste trabalho foi avaliado o desempenho da implementação em arquitetura NUMA. Máquinas NUMA têm a vantagem de agregar maior paralelismo ao adicionar mais processadores sem aumentar o gargalo de acesso ao barramento. Sua principal característica é o tempo não uniforme de acesso à memória. Nestes

experimentos utilizamos o processador Intel Xeon E5-4650 com 96 núcleos e 192 threads em *hyper threading*, os experimentos rodaram com cenários de até 512 threads, sendo assim os testes com 256 e 512 threads ultrapassam o limite de threads em *hyper threading* da máquina utilizada.

As figuras 13 e 14 apresentam os resultados com tempo de execução em segundos para os benchmarks testados utilizando todas configurações heurísticas para os cenários de 1 a 512 threads. Temos como base de comparação os resultados obtidos com a TinySTM. Para estes experimentos o LTMS apresentou na maioria dos casos um melhor tempo de execução quando comparado a TinySTM. O melhor ganho de desempenho obtido foi para o experimento Intruder de configuração Latency-Sequential com 512 threads que obteve 96% de redução em seu tempo de execução.

O experimento Bayes 13a apresentou melhor tempo de execução para a TinySTM que o escalonador LSTM para maioria dos cenários de threads, o teste com 4 threads com a configuração Threshold-Sequential apresentou o pior resultado obtendo 164.14% de tempo acima da TinySTM. O LTMS também apresentou algumas reduções no tempo de execução quando comparado a TinySTM, o teste Threshold-Chunks obteve uma redução de 46% no tempo de execução.

No experimento Intruder 13b o LTMS apresentou melhor tempo de execução para todos os cenários a partir de 2 threads, obtendo uma redução de até 96% do tempo de execução que ocorreu em Latency-Sequential com 512 threads, e o menor ganho de desempenho foi de 23% no teste Threshold-Chunks para 4 threads. Todos os resultados do Intruder para uma thread apresentaram menor tempo de execução, obtendo no pior cenário, Threshold-Chunks, um aumento de 24%.

O experimento Kmeans 13c apresentou para maioria dos cenários de threads um decremento no tempo de execução do LTMS em relação a TinySTM. O pior resultado ocorreu para o teste Threshold-Chunks com 512 threads e apresentou um aumento de 45% no tempo de execução, e o melhor resultado ocorreu para o teste Threshold-Sequential com 8 threads e apresentou um decremento de 80% no tempo de execução.

No experimento Labyrinth 14a o LTMS apresentou para maioria dos cenários um decremento no tempo de execução. O melhor tempo de execução em relação a TinySTM pode ser observado no teste Latency-Sequential para 512 threads, onde é observado uma redução de 54% do tempo de execução. O pior cenário para o LTMS apresenta um aumento de 29% que foi observado no teste Latency-Chunks 16 threads.

No experimento Vacation 14b foi observado uma redução de 81% do tempo de execução para o teste Latency-Sequential com 512 threads, a maioria dos cenários de threads do Vacation apresentam uma redução no tempo de execução do LTMS quando comparados ao TinySTM, porém, alguns cenários apresentaram um aumento

no tempo de execução, o pior cenário possui um aumento de 64% do tempo e pode ser visualizado no teste Latency-Chunks com 128 threads.

O experimento Yada 14c também apresentou para maioria dos testes uma redução do tempo de execução do LTMS em comparação a TinySTM. O LTMS obteve uma redução de até 92% do tempo de execução para o teste Latency-Chunks com 512 threads. O benchmark também apresentou uma redução do tempo de execução de 70% para o teste Threshold-Chunks com 16 threads.

Os resultados apresentam para maioria dos benchmarks um resultado melhor de desempenho utilizando o escalonador proposto neste trabalho. Quando ultrapassado o número de threads hyper threading disponíveis na arquitetura utilizada, os únicos benchmarks que não obtiveram resultados de tempo abaixo da TinySTM foram os benchmarks Bayes e Kmeans... **To do (??)** ... Os demais benchmarks apresentam uma redução significativa em seu tempo de execução para todos cenários de threads, sendo que demonstram maior expressividade para os cenários acima de 256 threads, aqui representando o cenário superior as threads disponíveis na arquitetura.

As figuras 15 e 16 apresentam os resultados de quantidades de aborts obtidos nos experimentos descritos acima. Tendo como base de comparação a biblioteca TinySTM, o LTMS apresentou para maioria dos benchmarks um desempenho superior, reduzindo no melhor caso até 99% do número de aborts existentes nas aplicações, esse resultado pode ser observado nos benchmarks Intruder, Kmeans, Vacation e Yada. Os benchmarks Intruder e Yada utilizando Latency-Sequential para 512 threads apresentaram ganho de 99,99% na redução de aborts. Os benchmarks Kmeans utilizando Threshold-Sequential e Vacations utilizando Latency-Chunks reduziram respectivamente 99,88% e 99,84% dos aborts para o cenario de 512 threads.

O experimento Bayes 15a com o LTMS apresenta para maioria dos testes um aumento no número de aborts. O teste Latency-Chunks com 4 threads apresentou um aumento de 18% no número de aborts. Por outro lado, o escalonador obteve para Latency-Sequential com 2 threads uma redução de 34% na média dos aborts.

No experimento Intruder 15b foi obtido ganho de desempenho para todos cenários de threads e configurações do escalonador. O melhor desempenho pode ser observado para Latency-Sequential com 512 threads, no qual foi obtido 99% de redução nos aborts. A menor redução nos aborts ocorreu para Threshold-Sequential com 2 threads, onde foi observado a redução de 89% dos aborts.

O experimento Kmeans 15c assim como Intruder, apresenta redução do número de aborts para todos os cenários e configurações. Podemos observar uma redução de até 99% para Threshold-Sequential com 512 threads. O pior caso do benchmark Kmeans apresentou uma redução de 71% dos aborts e pode ser observado no Latency-Sequential com 32 threads.

No experimento Labyrinth 16a apresentou para maioria dos cenários um decre-

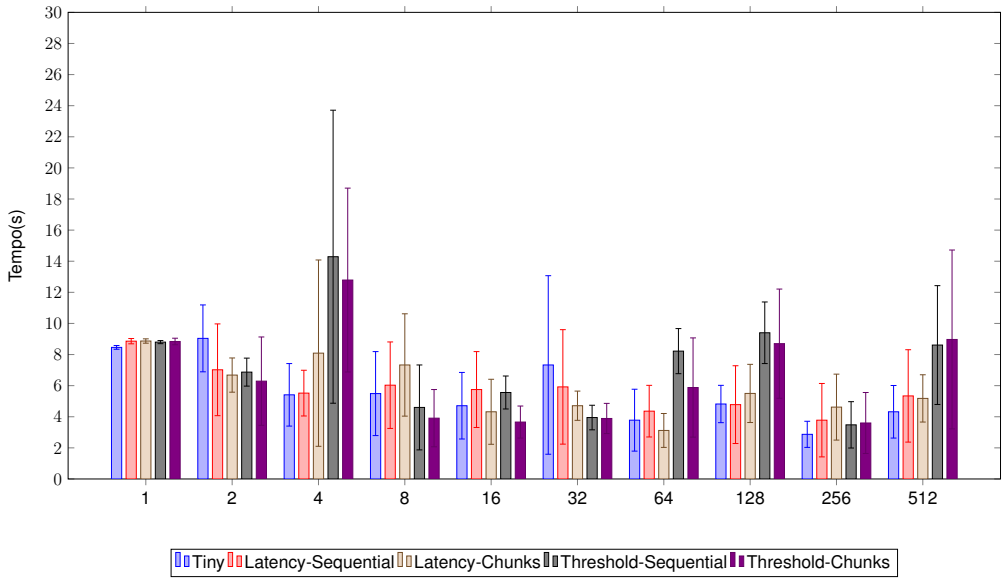
mento do número de aborts. O LTMS obteve até 54% de redução dos aborts, este resultado foi obtido no experimento Latency-Chunks com 512 threads. O LTMS também obteve um aumento de 12% dos aborts em relação a TinySTM para o teste Threshold-Chunks com 16 threads.

No experimento Vacation 16b foi observada a redução do número de aborts para todos os testes executados. O melhor valor obtido no LTMS quando comparado a TinySTM foi observado no Latency-Chunks com 512 threads, onde foi obtido 99% de redução nos aborts. Para Threshold-Chunks com 32 threads foi observado a menor redução do número de aborts com 27% de redução em relação a TinySTM.

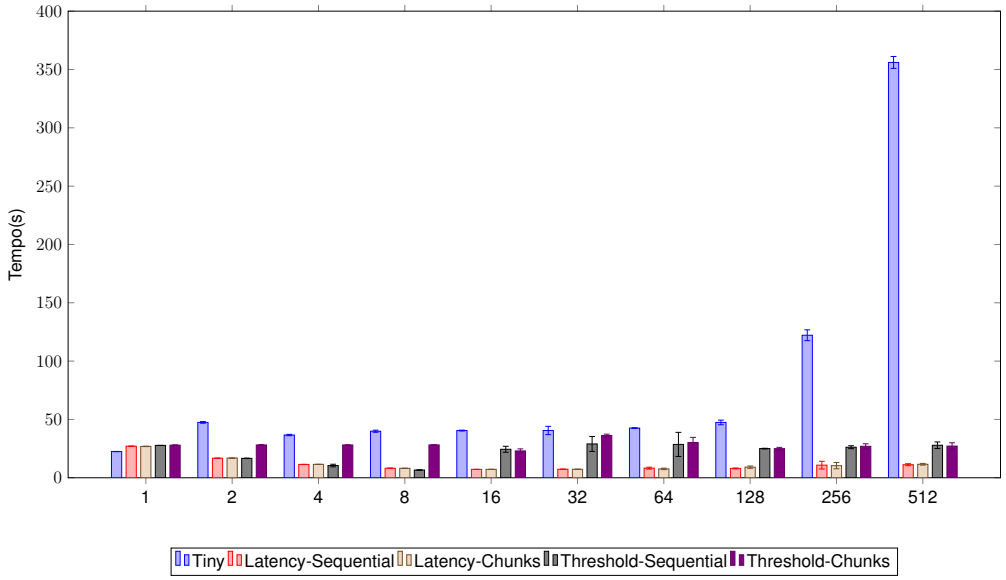
Por fim, o experimento Yada 16c também apresentou para todos os testes uma redução expressiva no número de aborts. Onde pode ser observado uma redução de 99% no teste Latency-Sequential com 512 threads. A menor redução de aborts apresentada no escalonador junto ao benchmark Yada foi de 97% e ocorreu com a configuração Latency-Chunks com 2 threads.

Os resultados apresentam para maioria dos experimentos uma redução significativa do número de abort quando utilizado o escalonador proposto. Quando ultrapassado o número de threads hyper threading disponíveis na arquitetura utilizada, o único benchmark que não obteve redução de aborts foi o Bayes. Os demais benchmarks ao ultrapassar o número de threads disponíveis na arquitetura apresentam uma redução significativa, chegando até 99% de redução dos aborts.

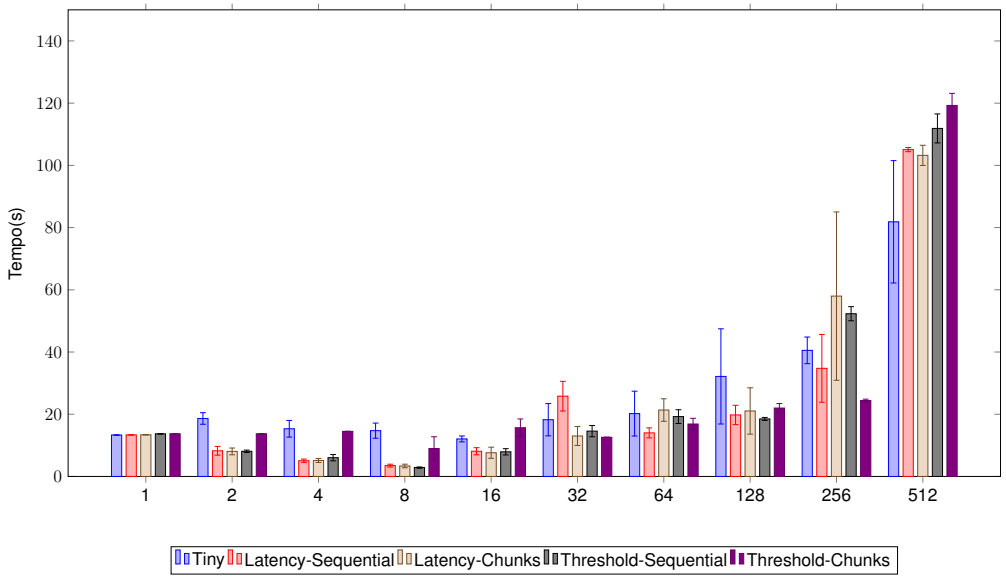
Para maioria dos benchmarks o tempo de execução e número de aborts obtiveram melhor desempenho utilizando o LTMS, estes resultados possuem em seu melhor cenário uma redução de 96% do tempo de execução e 99% do número de aborts. Porém, alguns benchmarks obtiveram resultados piores com o LTMS em diferentes configurações quando comparados a TinySTM. Isto mostra a importância de entendermos a aplicação. Ter um escalonador capaz de entender melhor as características da aplicação e conhecer a arquitetura pode como vimos na maioria dos benchmarks obter resultados expressivos.



(a) Bayes

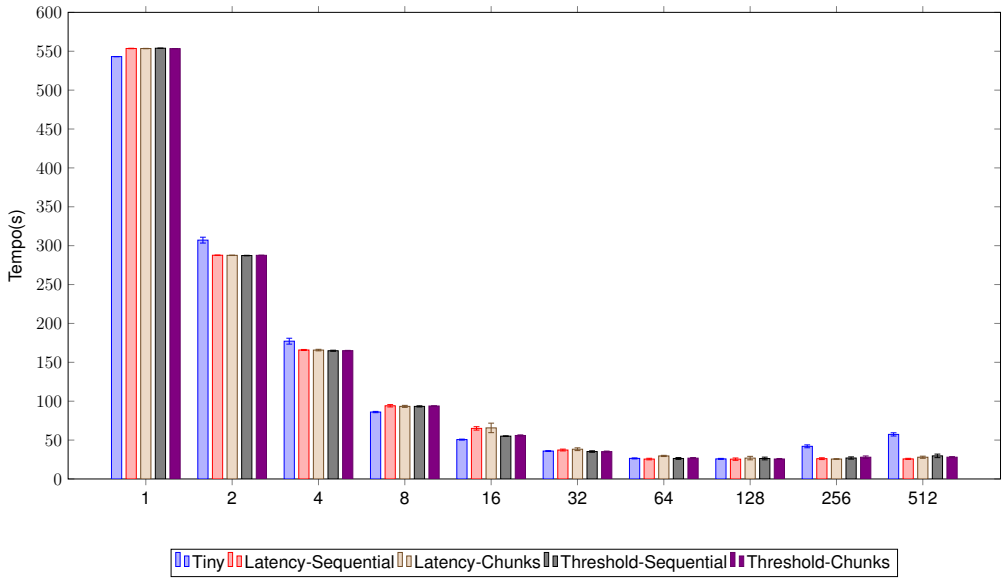


(b) Intruder

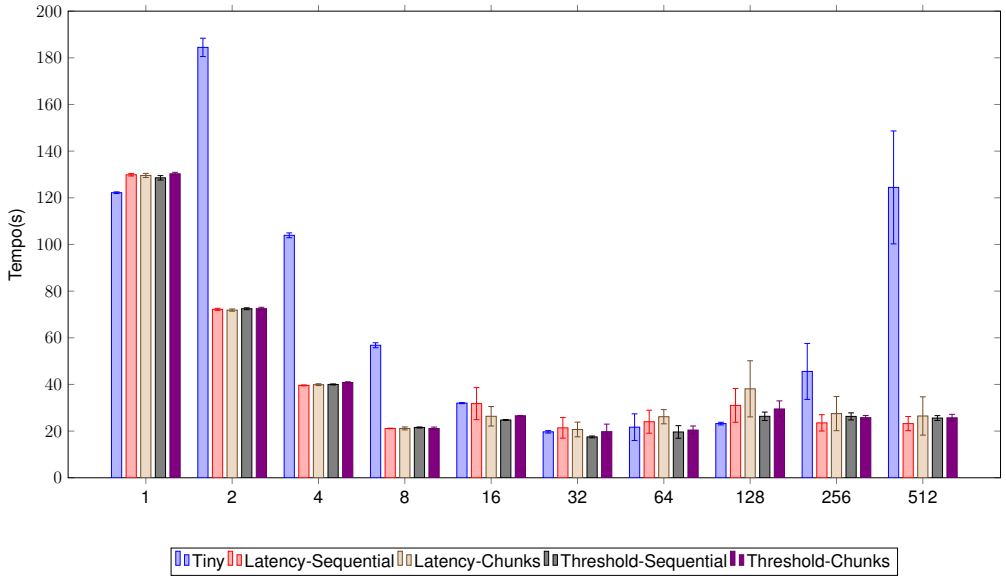


(c) Kmeans

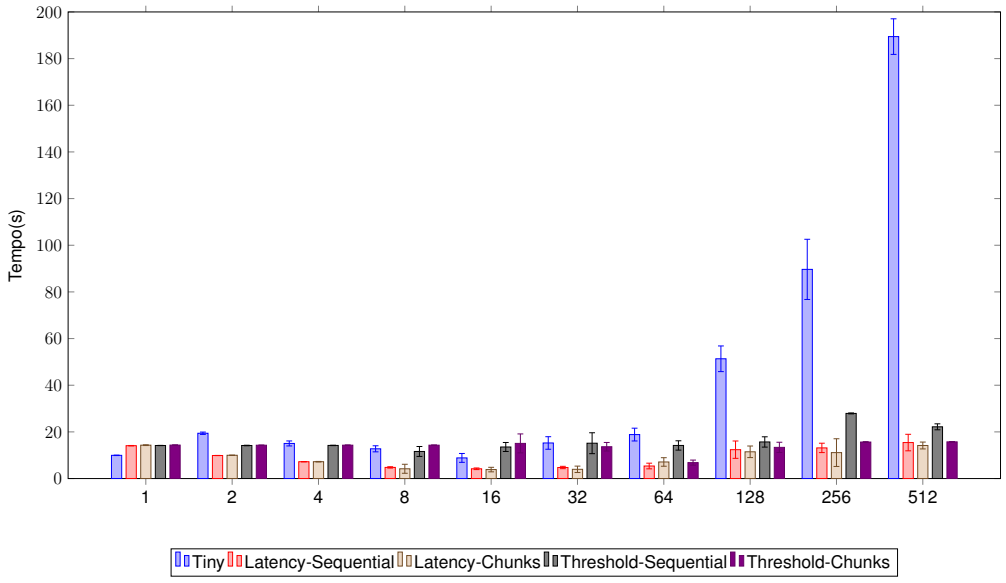
Figura 13 – Tempo de execução (s) em NUMA variando o número de *threads*.



(a) Labyrinth

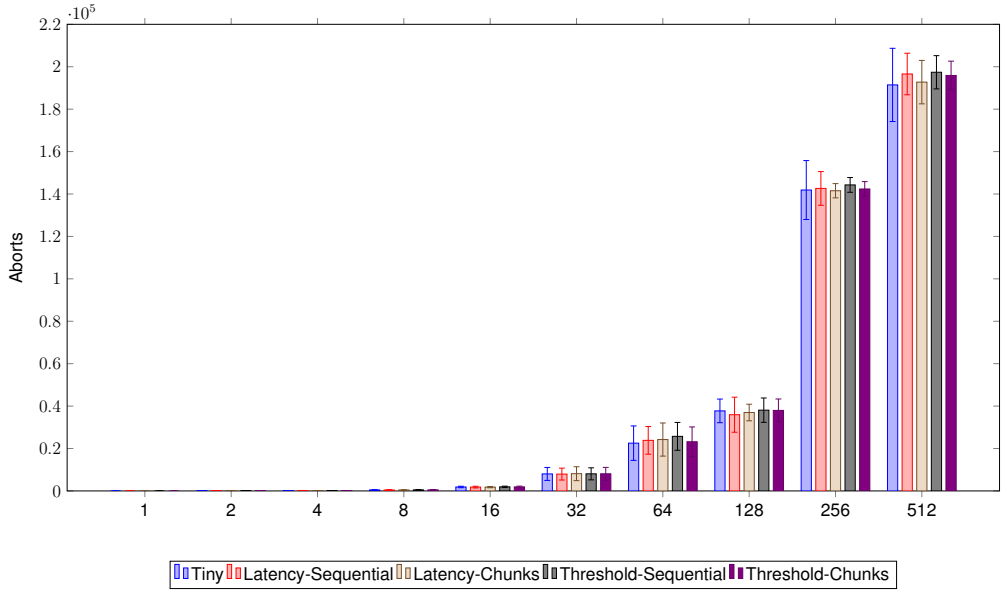


(b) Vacation

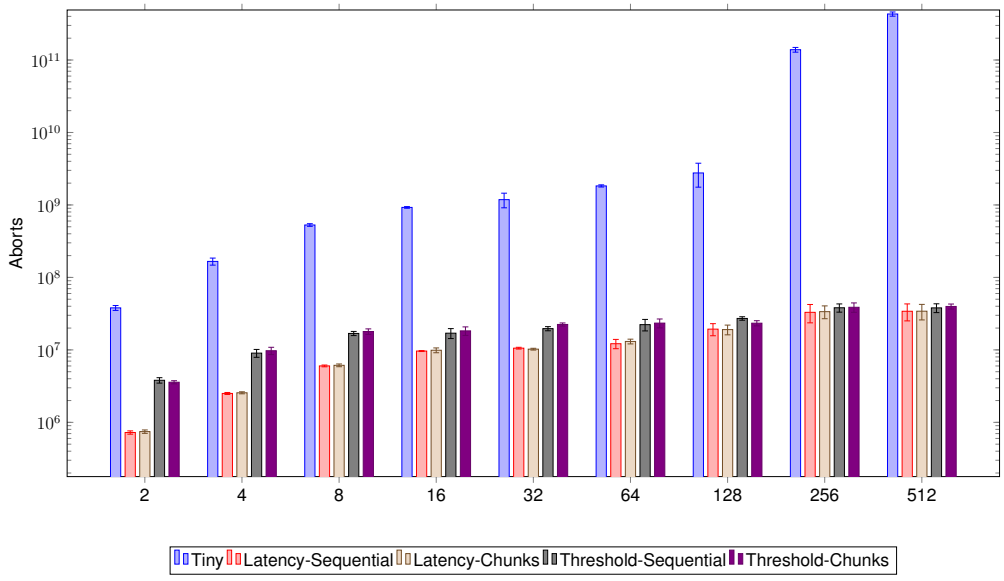


(c) Yada

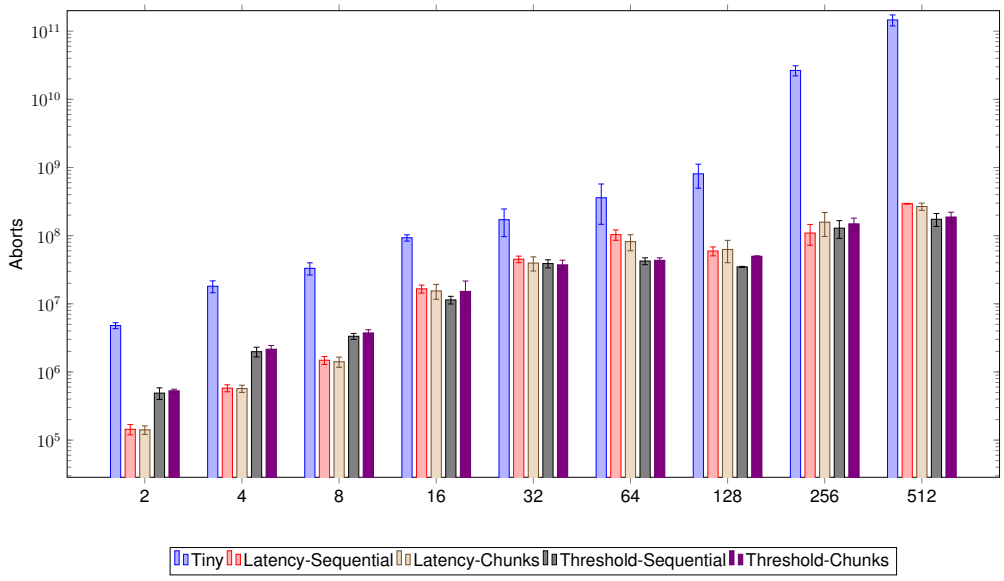
Figura 14 – Tempo de execução (s) em NUMA variando o número de *threads*.



(a) Bayes

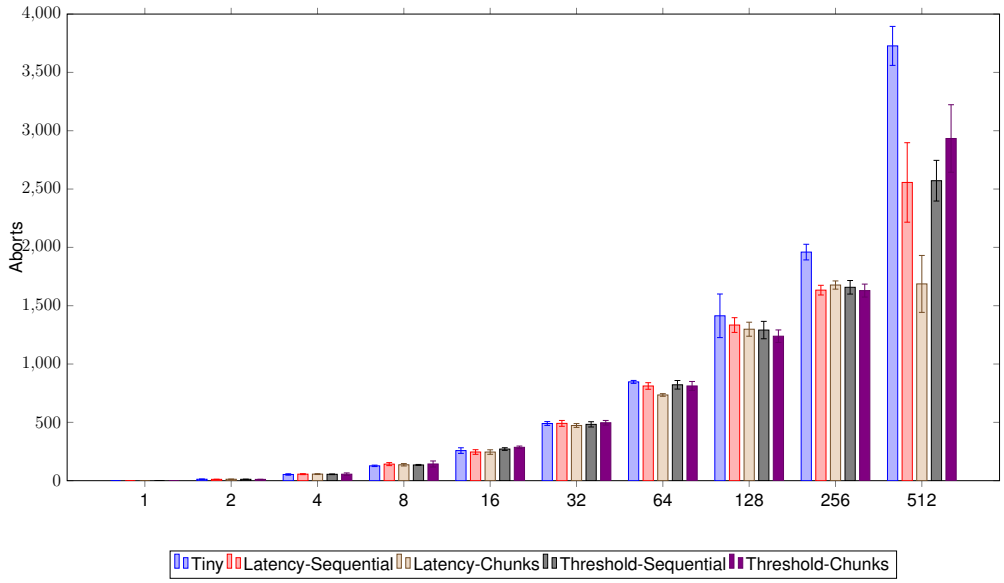


(b) Intruder

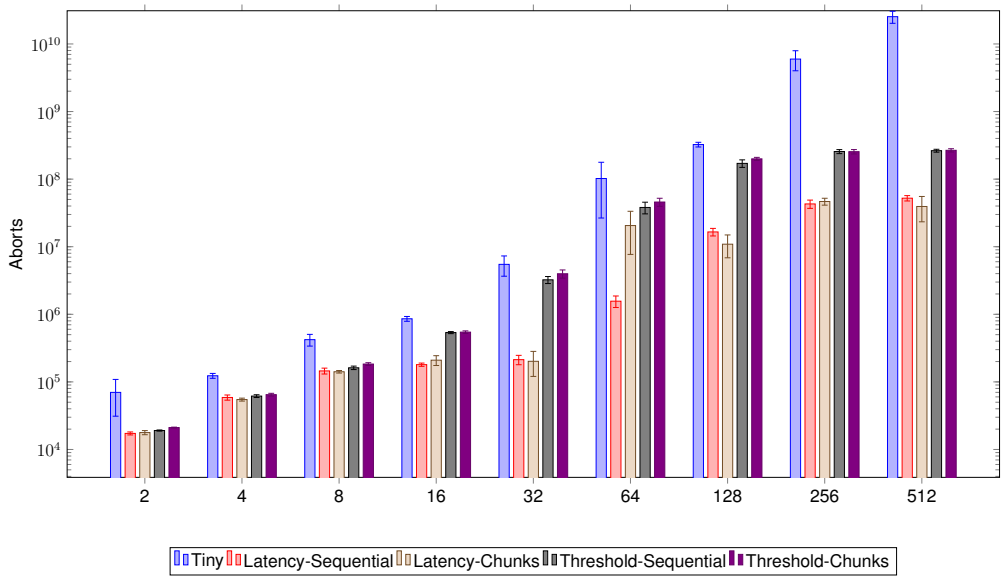


(c) Kmeans

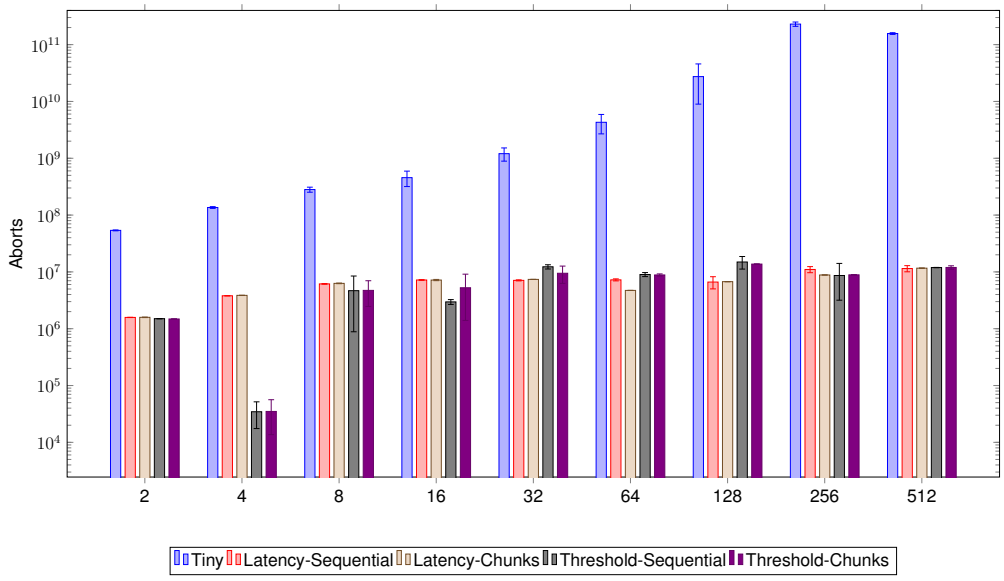
Figura 15 – Aborts em NUMA variando o número de *threads*.



(a) Labyrinth



(b) Vacation



(c) Yada

Figura 16 – Aborts em NUMA variando o número de *threads*.

6 CONCLUSÃO

Memória transacional (TM) fornece uma abstração para sincronização de threads em programas paralelos. Estas podem ser implementadas em hardware (HTM), software (STM) ou de forma híbrida com hardware e software. Este trabalho se concentrou em aplicações de STM. Muitos estudos de STM apresentam escalonadores transacionais que focam em reduzir o número de conflitos por meio da serialização das transações, reduzindo as threads ativas na aplicação ou bloqueando as transações conflitantes. Reduzir o número de conflitos se mostra eficiente para melhorar o desempenho, mas em arquiteturas multicore atuais com hierarquias de memória complexas também é importante considerar onde a posição de memória utilizada está localizada e por qual núcleo ela é acessada.

O entendimento da arquitetura utilizada, threads e dados é importante para o desempenho, melhorando a localidade e reduzindo a latência dos acessos à memória. As aplicações que utilizam STM oferecem oportunidades interessantes para entendimento da arquitetura e aplicação, pois em tempo de execução o STM fornece informações precisas sobre as áreas de memória que são compartilhadas entre threads, seus respectivos endereços e a intensidade com que cada endereço é acessado pelas threads. A partir do mapeamento destas informações este trabalho desenvolveu um escalonador NUMA-Aware que permite a migração de threads entre as filas de execução.

A principal contribuição desta dissertação é um escalonador de STM, entitulado *LTMS* e apresentado em 4, que adquira conhecimento sobre a arquitetura utilizada e sua aplicação, este conhecimento é gerado em tempo de execução por meio da coleta de dados das threads e suas transações. Este escalonador fornece um mecanismo de distribuição inicial de threads, criação de filas, e migração de threads conflitantes entre as filas existentes. Este mecanismo busca otimizar o desempenho de aplicações paralelas, reduzindo conflitos repetidos por meio da migração de threads agrupandoas as threads conflitantes na mesma fila. Para o desenvolvimento deste escalonador foram apresentadas três etapas que têm como contribuições individuais do trabalho.

A etapa de inicialização contribui com um sistema de filas individuais para os núcleos, apresentado em 4.2.1, na qual para executar a distribuição das threads entre estas filas podemos utilizar diferentes heurísticas para estudos, este trabalho apresentou duas heurísticas de distribuições. Estas heurísticas buscam distribuir inicialmente estas threads entre as filas disponíveis para entender o comportamento do número inicial de conflitos existentes na aplicação e comportamento do acesso à memória.

Em tempo de execução o escalonador contribui com um mecanismo para coleta de dados da STM, apresentado em 4.2.2, onde são montados duas matrizes uma de comunicação e uma de endereços. A matriz de comunicação fornece insumos sobre a quantidade de acessos aos endereços em comum entre threads. A matriz de endereços apresenta o endereço em comum mais acessado entre duas threads. Além destas matrizes, são coletadas informações sobre a latência de acesso à memória entre os nodos disponíveis na arquitetura e a quantidade de abortes e commits que ocorreram em cada thread. Estas informações são insumos para compreender a arquitetura e a aplicação e são fundamentais para tomada de decisão na etapa de migração de threads do escalonador.

A última contribuição deste trabalho está em um sistema de migração de threads, apresentado em 4.2.3, que é ativado apenas após a ocorrência de um abort da aplicação. O escalonador identifica a thread que gerou o abort, e com base nas informações previamente coletadas decide identifica para qual fila deve migrar está thread. O escalonador tem como base migrar a thread para uma fila na qual exista uma thread com maior número de acessos em comum à memória. Para tomada de decisão, se uma thread deve ou não ser migrada, é possível utilizar diferentes heurísticas para estudo, nesta dissertação foram utilizadas duas heurísticas. A primeira baseia se na relação entre abort e commit, quando esta relação fica acima de um valor pré definido temos um indicativo de uma aplicação com alto índice de conflitos e o escalonador efetua a migração com intuito de reduzir os conflitos na aplicação, para identificar o valor limiar utilizado neste trabalho foram executados testes onde chegamos ao limiar de 0.8, que indica que com 80% de abortes a thread deve ser migrada. A segunda heurística utiliza o custo da latência no acesso à memória como parâmetro para efetuar a migração, onde se a fila atual que o thread executa possuir latência maior que a fila para qual desejamos migrar o escalonador efetua a migração, está heurística busca reduzir o custo de latencia gasto na aplicação.

O escalonador foi desenvolvido sobre a biblioteca de STM TinySTM, e foram executados testes junto do conjunto de benchmarks STMAP, os experimentos foram rodados com o escalonador com ambas configurações de heurísticas desenvolvidas, como apresentado em 5, e comparados com a biblioteca original de TinySTM. Concluímos que as aplicações utilizando LTMS apresentam para maioria dos cenários e configurações uma redução no tempo de execução e número de aborts. Sendo que o LTMS

quando comparado a TinySTM obteve uma redução de até 96% do tempo de execução e 99% do número de aborts com benchmark Intruder utilizando configuração Latency-Sequential.

6.1 Trabalhos futuros

A pesquisa apresentada neste trabalho pode ser estendida das seguintes formas:

- *Novas heurísticas de distribuição*: O LTMS permite explorar distintas heurísticas de distribuição de threads ao inicializar a aplicação. Com isso é importante avançarmos nos estudos para explorar diferentes formas de distribuição e compará-las. As distintas características entre as heurísticas de distribuição podem ser exploradas em trabalhos futuros.
- *Heurística de migração híbrida*: O LTMS permite aplicar duas heurísticas de migração, uma com foco no índice de contenção e outra com foco na latência de acesso à memória, mas o escalonador permite que estas heurísticas possam ser utilizadas juntas para aperfeiçoar o sistema de migração, buscando a redução da contenção e otimização do acesso à memória para reduzir a latência da aplicação.
- *Impacto energético dos escalonadors de STM*: Os trabalhos atuais focam no impacto que os escalonadores possuem sobre o desempenho de tempo e redução de conflitos. Um aspecto a ser abordado em trabalhos futuros pode ser o impacto que escalonador com consciência da arquitetura, como o LTMS, possuem em relação ao custo energético em relação a outros escalonadores.

O escalonado LTMS, desenvolvido nesta dissertação, pode ser estendido permitindo que estas características sejam avaliadas em trabalhos futuros. Assim, podemos seguir com o desenvolvimento do LTMS contribuindo e explorando outros aspectos da área.

REFERÊNCIAS

BALDASSIN, A. J. **Explorando Memória Transacional em Software nos Contextos de Arquiteturas Assimétricas, Jogos Computacionais e Consumo de Energia**. 2009. Dissertação de Doutorado — Universidade Estadual de Campinas.

BANDEIRA, R. de Leão. **Compilador para a linguagem CMTJava**. 2010. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) — Universidade Federal de Pelotas.

CALCIU, I.; SEN, S.; BALAKRISHNAN, M.; AGUILERA, M. K. How to Implement Any Concurrent Data Structure for Modern Servers. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.51, n.1, p.24–32, Sept. 2017.

DI SANZO, P. Analysis, classification and comparison of scheduling techniques for software transactional memories. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.28, n.12, p.3356–3373, 2017.

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional Locking II. In: DISC 2006, 2006. **Anais...** [S.l.: s.n.], 2006. p.194–208.

DOLEV, S.; HENDLER, D.; SUISSA, A. CAR-STM: Scheduling-based Collision Avoidance and Resolution for Software Transactional Memory. In: TWENTY-SEVENTH ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.125–134. (PODC '08).

DRAGOJEVIć, A.; GUERRAOUI, R.; SINGH, A. V.; SINGH, V. Preventing Versus Curing: Avoiding Conflicts in Transactional Memories. In: ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 28., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.7–16. (PODC '09).

FAVARETTO, R. M. **Escalonamento dinâmico em nível aplicativo sensível à arquitetura e às dependências de dados entre as tarefas**. 2014. Dissertação de Mestrado — PPGC/UFPEL, Pelotas/RS.

FELBER, P.; FETZER, C.; RIEGEL, T. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: PPOPP '08: PROC. OF THE 13TH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.237–246.

HAAGDORENS, B.; VERMEIREN, T.; GOOSSENS, M. Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading. In: INFORMATION SECURITY APPLICATIONS, 2005. **Anais...** [S.l.: s.n.], 2005. p.188–203. (Lecture Notes in Computer Science (LNCS), v.3325).

HARRIS, T.; LARUS, J.; RAJWAR, R. Transactional Memory, 2nd edition. **Synthesis Lectures on Computer Architecture**, [S.l.], v.5, n.1, p.1–263, 2010.

HERLIHY, M.; ELIOT, J.; MOSS, B. Transactional Memory: Architectural Support for Lock-Free Data Structures. In: PROC. OF THE 20TH ANNUAL INTL. SYMPOSIUM ON COMPUTER ARCHITECTURE, 1993. **Anais...** [S.l.: s.n.], 1993. p.289–300.

MINH, C. C.; CHUNG, J.; KOZYRAKIS, C.; OLUKOTUN, K. STAMP: Stanford Transactional Applications for Multi-Processing. In: WORKLOAD CHARACTERIZATION, 2008. IISWC 2008. IEEE INTERNATIONAL SYMPOSIUM ON, 2008. **Anais...** [S.l.: s.n.], 2008. p.35–46.

MORESHET, T.; BAHAR, R. I.; HERLIHY, M. Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks. In: WORKSHOP ON MEMORY PERFORMANCE ISSUES, 2006. **Proceedings...** [S.l.: s.n.], 2006.

NICÁCIO, D.; BALDASSIN, A.; ARAÚJO, G. Transaction Scheduling Using Dynamic Conflict Avoidance. **International Journal of Parallel Programming**, [S.l.], v.41, n.1, p.89–110, 2012.

PASQUALIN, D. P.; DIENER, M.; DU BOIS, A. R.; PILLA, M. L. Online Sharing-Aware Thread Mapping in Software Transactional Memory. In: IEEE 32ND INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p.35–42.

PASQUALIN, D. P.; DIENER, M.; DU BOIS, A. R.; PILLA, M. L. Thread affinity in software transactional memory. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED COMPUTING (ISPDC), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p.180–187.

RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. **Memorias Transacionais**: Uma Nova Alternativa para Programação Concorrente. [S.l.]: In Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing, 2007.

RITO, H.; CACHOPO, J. Adaptive transaction scheduling for mixed transactional workloads. **Parallel Computing**, [S.l.], v.41, p.31–49, 2015.

RUPPERT, J. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. **J. Algorithms**, Duluth, MN, USA, v.18, n.3, p.548–585, May 1995.

YOO, R. M.; LEE, H.-H. S. Adaptive transaction scheduling for transactional memory systems. In: PARALLELISM IN ALGORITHMS AND ARCHITECTURES, 2008. **Proceedings...** [S.l.: s.n.], 2008. p.169–178.