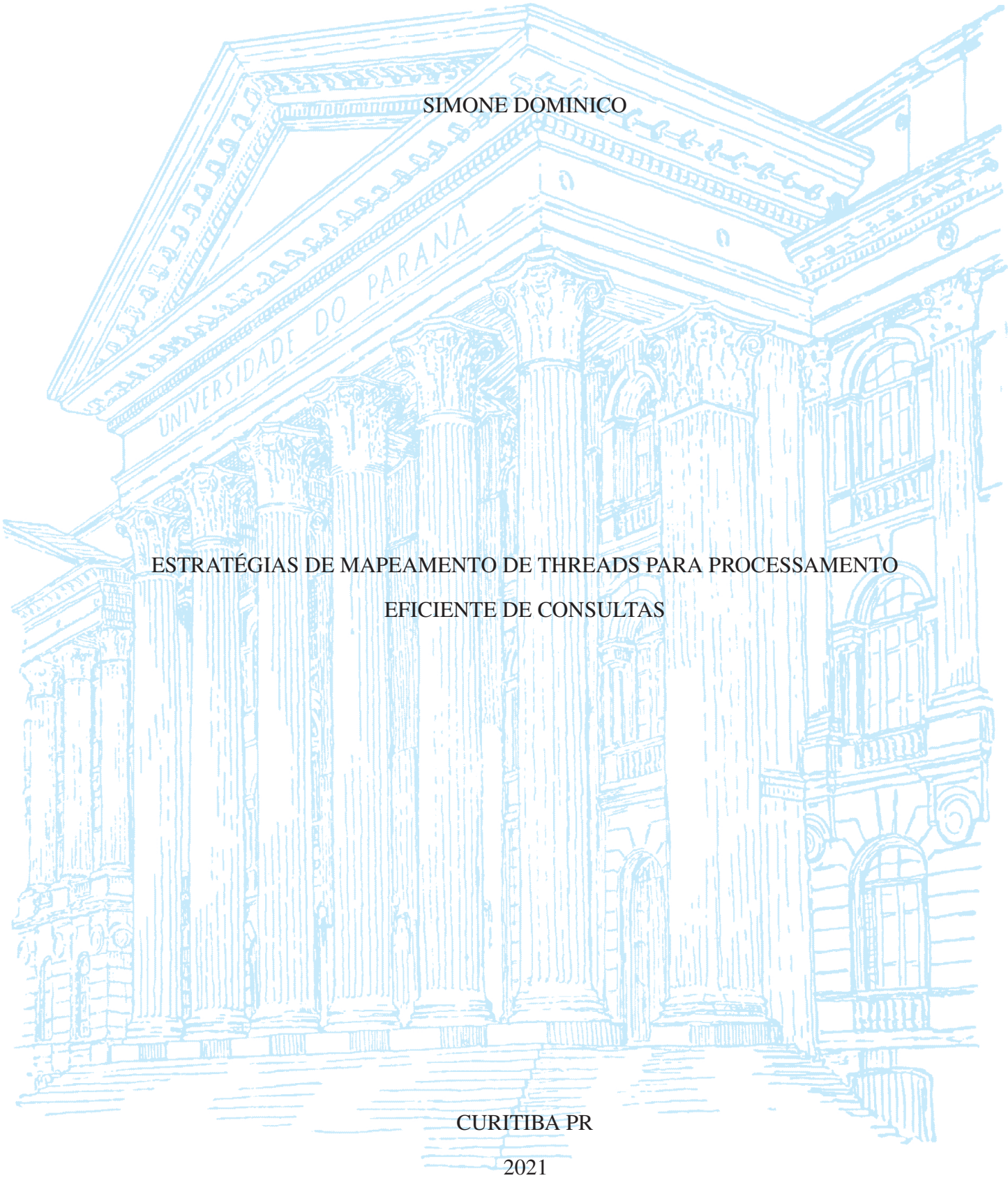
UNIVERSIDADE FEDERAL DO PARANÁ

SIMONE DOMINICO



ESTRATÉGIAS DE MAPEAMENTO DE THREADS PARA PROCESSAMENTO EFICIENTE DE CONSULTAS

CURITIBA PR

2021

SIMONE DOMINICO

ESTRATÉGIAS DE MAPEAMENTO DE THREADS PARA PROCESSAMENTO EFICIENTE DE CONSULTAS

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Eduardo Cunha de Almeida.

Coorientador: Marco Antonio Zanata Alves.

CURITIBA PR

2021

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

D671e Dominico, Simone

Estratégias de mapeamento de Threads para processamento eficiente de consultas [recurso eletrônico] / Simone Dominico - Curitiba, 2021.

Tese (doutorado) - Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Eduardo Cunha de Almeida
Coorientador: Prof. Dr. Marco Antonio Zanata Alves

1. Arquitetura de redes de computador. 2. Banco de dados. 3. Threads (Programa de computador). I. Universidade Federal do Paraná. II. Almeida, Eduardo Cunha de. III. Alves, Marco Antonio Zanata. VI. Título.

CDD 005.43

Bibliotecário: Nilson Carlos Vieira Junior CRB9/1797

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **SIMONE DOMINICO** intitulada: **Estratégias de Mapeamento de Threads para Processamento Eficiente de Consultas**, sob orientação do Prof. Dr. EDUARDO CUNHA DE ALMEIDA, que após terem inquirido a aluna e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutora está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 15 de Fevereiro de 2022.

Assinatura Eletrônica
23/02/2022 14:01:05.0
EDUARDO CUNHA DE ALMEIDA
Presidente da Banca Examinadora

Assinatura Eletrônica
22/02/2022 17:05:00.0
MATTHIAS DIENER
Avaliador Externo (UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN)

Assinatura Eletrônica
25/02/2022 09:14:55.0
LUIS CARLOS ERPEN DE BONA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
22/02/2022 16:49:52.0
FABIO ANDRE MACHADO PORTO
Avaliador Externo (LABORATÓRIO NACIONAL DE COMPUTAÇÃO CIENTÍFICA)

Dedico este trabalho ao meu marido e companheiro Tiago de Lima Begnini (in memorian) que me apoiou em todos os momentos durante essa jornada. Infelizmente, ele não conseguiu ver pessoalmente a conclusão dessa tese porque não esta mais entre nós, mas continua sendo a minha força nessa vida. Nossos sonhos continuam me ajudando a persistir e nunca desistir. Espero que me perdoe por sacrificar tanto do nosso tempo.

AGRADECIMENTOS

Escrever os agradecimentos deve ser uma das partes mais difíceis. Ao longo do desenvolvimento desta tese passamos por tantas emoções que é difícil de quantificar a importância de cada um para finalizar mais uma etapa. A única coisa que tenho certeza é que essa tese não seria possível sem todos o apoio, paciência, conhecimento, ensinamentos e por todo o suporte que recebi dos meus orientadores, Eduardo Cunha de Almeida e Marco Antonio Zanata Alves.

Gostaria de agradecer aos colegas de laboratório de banco de dados (LBD), pelas longas conversas sobre pesquisas, dúvidas e trajetória. Em especial ao Edson Lucas Ramiro Filho, Fabíola Santore, Tiago Rodrigo Kepe e Eduardo H. Monteiro Pena. Agradeço aos colegas do laboratório HiPES, que me acolheram e me aceitaram. Além disso, pelas longas conversas que tínhamos nas reuniões do grupo HiPES, pela sinceridade, parceira ao longo da caminhada, pelo apoio emocional durante a pandemia, poderia citar vários nomes em especial, porém considero todos e todas como parte essencial da minha formação. Agradeço também a amizade, paciência e tempo investido na correção desta tese, pela pessoa sensacional que tive o prazer de conhecer, obrigada Marisa Sel Franco por me auxiliar nesse processo.

Meus sinceros agradecimentos ao PPGINF, incluindo todas a equipe de professores e equipe administrativa. Eu gostaria de agradecer a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e ao Serrapilheira pelo suporte financeiro para esta pesquisa. Também, gostaria de agradecer a equipe do O Parque Computacional de Alto Desempenho (PCAD) por disponibilizar a infraestrutura para desenvolvimento de alguns experimentos desta tese.

Agradeço a Deus pela luz e força que me concedeu, por me guiar e me dar tranquilidade para seguir com os meus objetivos e não desanimar com as dificuldades. Agradeço a minha família pelo suporte, minha mãe preocupada com minha saúde, que mesmo sem entender a importância me apoiou.

Para finalizar, a parte mais importante e essencial meu amado esposo Tiago de Lima Begnini, quando eu queria desistir você não deixou, me apoiou incondicionalmente, mesmo isso significando o esforço e sacrifício de nós dois. Agradeço a minha filha Agatha que cresceu ao lado das minhas pesquisas, que enquanto eu evoluía na minha carreira ela crescia e absorvia muitos das minhas dúvidas, vitórias e frustrações. Ao meu filho, Murilo, que, mesmo pequeno entendia a importância desta etapa e aceitou. A vocês três sou grata por estarem ao meu lado, por se doarem junto comigo, por serem parte desta tese tão intensamente que sem vocês, não seria possível.

RESUMO

Os Sistemas Gerenciadores de Banco de Dados (SGBDs) modernos usam a execução paralela de consultas para cumprir requisitos de desempenho importantes, como baixo tempo de resposta e grande vazão no processamento de consultas. Contudo, cumprir requisitos de desempenho com o aumento exponencial dos volumes de dados gerados na atual era digital tem sido um constante desafio. Neste sentido, o desenvolvimento dos SGBDs requer o envolvimento com os avanços nas arquiteturas de *hardware* e sistemas operacionais (SO) para aproveitar de forma eficiente os múltiplos recursos de *hardware* disponíveis, como processadores, memória e discos. Nesta tese exploramos a execução de consultas em arquitetura de acesso não-uniforme à memória (*Non-uniform memory access* (NUMA)) que tem potencial para oferecer alta vazão através de vários chips contendo unidades de processamento de múltiplos núcleos agrupados em nós de processamento. No entanto, os SGBDs modernos não exploram o *hardware* NUMA em todo o seu potencial deixando para os SOs o trabalho de alocação de *threads* geradas pelas consultas. A alocação de *threads* dos SOs acarreta em movimento de dados ineficiente entre os nós NUMA dado que os SOs são projetados para cargas de trabalho de propósito geral e políticas de balanceamento de carga. Esta tese apresenta dois mecanismos de alocação de *threads* de execução de consultas. O primeiro mecanismo é direcionado para os SGBDs relacionais. O mecanismo proposto utiliza um *pipeline* de regra-condição-ação que, por meio de contadores de *hardware*, encontra prontamente o número ideal local de núcleos. Esse mecanismo usa uma fila de prioridade que rastreia o histórico do espaço de endereço de memória usado pelas *threads* da execução de consultas para decidir sobre a alocação/liberação de núcleos e sua distribuição entre os nós NUMA e, com isso, diminui os acessos remotos à memória ao mesmo tempo em que aumenta consideravelmente a vazão e o *speedup* máximo. O segundo mecanismo considera os SGBDs multidimensionais. Este mecanismo controla a alocação de *threads* por meio de um algoritmo de jogo multi-agente. Nesse contexto, as *threads* são agentes tomadores de decisão e escolhem a melhor alocação baseadas em informações de contadores de *hardware* e padrões de acesso à memória com redução dos acessos remotos e aumento na economia de energia. Em resumo, estes mecanismos permitem que SGBDs relacionais e multidimensionais aproveitem os recursos computacionais oferecidos por máquinas NUMA melhorando sensivelmente os tempos de resposta e vazão na execução paralela de consultas.

Palavras-chave: Banco de Dados; Execução Paralela de Consultas; Máquinas NUMA; Mapeamento de Threads;

ABSTRACT

Modern Database Management Systems (DBMS) use parallel query execution to meet performance requirements, like low response time and high throughput in query processing. With the exponential increase in data volumes generated in the current digital age, meeting these performance requirements in query execution has been a constant challenge. In this sense, there is a dire need for co-design among DBMSs, hardware architectures and operating systems (OS) to efficiently take advantage of the multiple hardware resources, such as processors, memory and disks. NUMA hardware has the potential to provide high throughput across multiple chips containing multi-core processing units grouped into processing nodes. However, modern DBMSs do not exploit the NUMA hardware to its full potential, leaving the allocation of query threads to the OSs. The allocation of threads from OSs leads to inefficient data movement between NUMA nodes as OSs are designed for general purpose workloads and load balancing policies. This thesis presents two query processing thread allocation mechanisms. The first mechanism relational DBMSs. The proposed mechanism uses a rule-condition-action pipe that, through hardware counters, readily finds the local optimal number of cores. This mechanism uses a priority queue that tracks the history of the memory address space used by the query execution threads to decide on the allocation/release of cores and their distribution among the NUMA nodes and, with that, decreases the remote memory accesses while considerably increasing the throughput and the maximum speedup. The second mechanism aims multidimensional DBMSs. This mechanism controls the allocation of threads through a multi-agent game algorithm. In this context, threads are decision-making agents that choose the best allocation based on information from hardware counters and memory access patterns, reducing remote access and increasing energy savings. In summary, these mechanisms allow relational and multidimensional DBMS to take advantage of the computational resources offered by NUMA machines, significantly improving the response times and throughput in the parallel processing of queries.

Keywords: Databases; Parallel Query Execution; NUMA machines; Thread mapping;

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

AMD	*Advanced Micro Devices*
API	*Application Programming Interface*
CEP	Controle Estatístico de Processos
CPU	*Central Processing Unit*
DPU	*Data Processing Unit*
EDP	*Energy-delay product*
GRU	*Global Reference Unit*
HT	*HyperTransport*
IMC	*Integrated Memory Controller*
LIC	Limite inferior de Controle
LLC	*Last level cache*
LONC	*Local Optimum Number of Cores*
LSC	Limite Superior de Controle
MAL	*Monet Assembly Language*
MMU	*Memory Management Unit*
MPI	*Misses per Instruction*
NUMA	*Non-Uniform Memory Access*
OLAP	*Online Analytical Processing*
OLTP	*Online Transaction Processing*
PID	*Process Identifier*
PCM	*Intel Performance Counter Monitor*
PrT	*Rede de Petri Predicado/Transição*
QPI	*Intel QuickPath Interconnect*
SGBD	Sistema Gerenciador de Banco de Dados.
SGBDR	Sistema Gerenciador de Banco de Dados Relacional.
SGBD de matriz	Sistema Gerenciadores de banco de dados de matriz
SMP	*Symmetric Multi-Processing*
SO	Sistema Operacional
SQL	*Structured Query Language*
TID	*Thread Identifier*
TLB	*Translation Lookaside Buffer*
TPC-H	*Transaction Processing Performance Council H*
UFPR	Universidade Federal do Paraná
UMA	*Uniform memory access*
UPI	*Intel Ultra Path Interconnect*

LISTA DE SÍMBOLOS

γ_{node}	Vinculação de *threads* em um nó NUMA
ϵ	Perfil de posicionamento
δ	Conjunto de todos os perfis do jogo
$\bowtie$	Operador de junção
π	Operador de projeção
σ	Operador de seleção

SUMÁRIO

1 INTRODUÇÃO

Ao longo dos anos, o volume de dados vem crescendo exponencialmente com estimativas de volume de dados globais na ordem de 180 zettabytes [Holst 2021] para o ano 2025. Há inúmeras fontes geradoras de dados na atual era digital, como servidores *web*, internet das coisas (IoT), redes sociais, dados abertos, simulações científicas, exploração de dados, aprendizado de máquina e estruturas biológicas, para citar apenas alguns. Dada essa diversidade de fontes, os dados possuem formatos distintos – o que exige também diferentes modelos de armazenamento de dados. Manipular e gerenciar dados com formatos diversos é, portanto, uma necessidade do mundo atual.

As organizações enfrentam desafios para realizar o processamento, a análise e o armazenamento de dados de forma eficiente, que levam às constantes iniciativas de pesquisa e desenvolvimento dos SGBDs. Os SGBD tem um papel fundamental nas organizações, pois é formado por um conjunto de programas que visa armazenar e acessar os bancos de dados eficientemente [Elmasri e Navathe 2000].

Esta tese concentra-se no conjunto de programas responsável pela execução paralela de consultas em dois tipos de SGBDs: relacionais e multidimensionais (também chamados de matriz). Os SGBD relacionais destacam-se há quase 50 anos, sendo amplamente utilizados devido ao seu esquema de armazenamento eficiente. O modelo relacional reduz o espaço de armazenamento e facilita a manutenção dos dados devido à independência de dados e a representação dos relacionamentos. Ele permite ainda que os usuários combinem e busquem dados de maneira flexível através do nome das relações, tuplas e atributos. Além disso, oferece uma maneira consistente de acesso a todos os elementos de dados usando a SQL como linguagem padrão de consulta declarativa [Codd 1985].

Além dos tradicionais SGBD relacionais, os SGBDs multidimensionais estão sendo cada vez mais utilizados principalmente para manipular dados provenientes de simulações científicas, exploração de dados, aprendizado de máquina, estruturas biológicas entre outras aplicações [Baumann e Holsten 2011; Kim et al. 2016]. Os SGBDs multidimensionais implementam um modelo de dados de matriz multidimensional para atender às muitas aplicações que possuem grande consumo de dados, mas que não se encaixam no modelo relacional tradicional. As linguagens de consulta de matriz fornecem operações multidimensionais específicas, como operações de álgebra linear e geométrica: Fatias de dados, transposição de matriz, adição, subtração e matriz oposta.

De modo geral, as aplicações dependem do desempenho dos SGBDs para obter resultados rápidos nas operações de processamento de dados. A maioria dos SGBDs são baseados em uma arquitetura concebida na década de 1970, otimizada para gargalos de E/S de disco e para execução em computadores de CPU única. Os principais requisitos de desempenho dos SGBDs estão relacionados com a taxa de transferência, latência, tempo de resposta, vazão e escalabilidade.

Ao mesmo tempo, os avanços das arquiteturas de *hardware* transformaram as exigências de concepção e implementação dos SGBDs. Para atender aos requisitos de desempenho no processamento de consultas envolvendo grandes volumes de dados, tornou-se necessário explorar o paralelismo na etapa de execução das consultas. Assim, passou-se a dividir as operações de execução de uma consulta SQL em múltiplas *threads* para aproveitar os vários núcleos de processamento disponíveis (também chamado paralelismo intra-consulta). O processamento deve ser colaborativo, reduzindo o tempo de execução de operações essenciais e garantindo vazão para atender um número maior de cargas de trabalho por unidade de tempo.

As atuais arquiteturas de *hardware* multi-núcleos suportam a execução de mais de uma *thread* em simultâneo. Sendo assim, dividir as operações de consulta em múltiplas *threads* permite que partes dessa consulta sejam distribuídas em diferentes núcleos e executadas simultaneamente. Assim, torna-se possível aumentar o desempenho de SGBDs por meio do paralelismo de consulta. Há, porém, um descompasso nessa relação: a evolução dos *hardwares* não é acompanhada pelos SGBDs. Dominar o poder computacional fornecido pelas arquiteturas de *hardware* de processamento multi-núcleos é evidentemente uma tarefa difícil para os desenvolvedores de SGBDs.

Um aspecto importante nas arquiteturas de *hardware* multi-núcleo é o modelo de arquitetura de memória. Os *hardwares* multi-núcleos atuais utilizam arquitetura de computação NUMA. Nessas arquiteturas NUMA, o acesso à memória realizado pelos processadores é não-uniforme: possui latências distintas. Além disso, a arquitetura NUMA é formada por nós de processamento multi-núcleos com hierarquias de memória complexas compostas por vários níveis de *cache* e diferentes esquemas de compartilhamento entre os núcleos. O aumento no número de núcleos implica no aumento na complexidade do *hardware* e, consequentemente, traz a necessidade de adaptações por parte dos SGBDs para aproveitar eficientemente o potencial processamento paralelo.

No entanto, os atuais SGBDs executam múltiplas *threads* sem explorar todo o potencial da arquitetura NUMA. O impacto da movimentação de dados entre os nós NUMA é um grande desafio para os SGBDs *multi-threads*. Os recursos de *hardware* habitualmente são gerenciados pelo SO, que pouco conhece sobre os requisitos de desempenho dos SGBDs e da natureza dos dados processados. Consequentemente, o SO migra e interrompe *threads* em todos os nós NUMA, com o objetivo de manter o balanceamento de carga e gerenciar os objetivos globais dos servidores. As decisões do SO afetam diretamente o desempenho dos SGBDs de forma inconsciente devido à falta de comunicação entre ambos [Kiefer et al. 2013].

Como resultado, os SGBDs tornam-se vulneráveis às decisões do SO porque acabam utilizando políticas padrão de gerenciamento de *threads* que geram grandes movimentações de dados entre os nós NUMA. Como consequência deste gerenciamento de *threads*, há um aumento do tempo de resposta no processamento de consultas devido à latência adicional imposta para acessar dados em diferentes nós. Sendo assim, existe uma lacuna crescente entre o poder computacional da arquitetura NUMA e o gerenciamento das múltiplas *threads* utilizadas na execução de consultas.

Assim, a obtenção de um bom desempenho na arquitetura NUMA envolve o posicionamento cuidadoso de *threads* com objetivo de reduzir acessos que acontecem em diferentes nós NUMA chamados de acessos remotos. Dentre os aspectos importantes para explorar a arquitetura NUMA, devemos considerar: a redução dos acessos às memórias remotas, a contenção nos controladores de memória e nos links de interconexão, além do reconhecimento da hierarquia de *cache*. Neste contexto, a presente pesquisa estuda o desempenho de SGBDs relacionais e multidimensionais executando em arquiteturas NUMA. Esta tese propõe dois mecanismos de mapeamento de *threads* na execução de consultas que permitem a redução dos acessos à memória remota, melhorando sensivelmente o desempenho na execução da carga de trabalho e minimizando custos de energia.

1.1 PROBLEMA

A Figura 1.1 apresenta uma arquitetura NUMA: cada nó multi-núcleo possui sua própria memória local, acessível aos outros nós através de *links* de interconexão. A latência de acesso à memória varia conforme a distância entre o nó no qual o núcleo que está fazendo o acesso aos

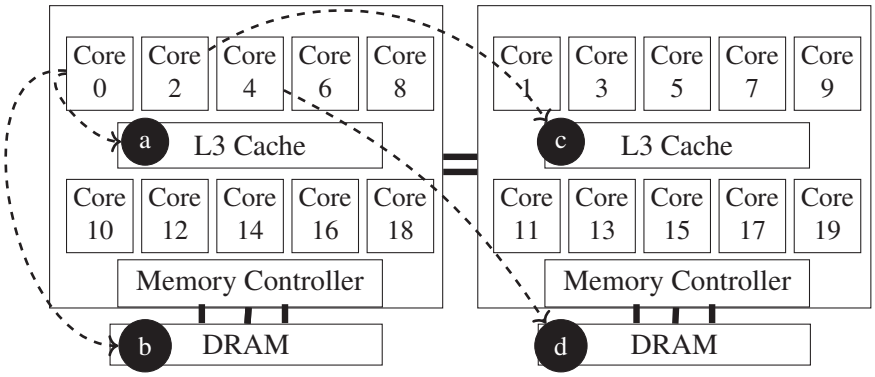


Figura 1.1: Exemplo de uma arquitetura NUMA com 2-nós baseada no Intel Xeon Silver 4114.

dados localiza-se e a memória que está sendo acessada. O acesso às memórias remotas impõe uma latência maior do que aquele feito às memórias locais devido ao uso da interconexão. Assim, o tempo de acesso à memória na arquitetura NUMA é não-uniforme. Como mostra a Figura 1.1, os acessos à memória *cache* (a) possuem um custo menor de latência, seguidos pelos acessos à memória local (DRAM) (b), à *cache* remota (c) e à memória remota (d).

Assumimos nesta tese o escalonamento de processos do SO Linux. O escalonador determina em qual nó NUMA o *kernel* irá alocar memória em relação às políticas de mapeamento e balanceamento de carga. O mapeamento de memória define o modo que os dados serão posicionados na memória. Ao iniciar uma *thread*, um SGBD não-consciente da arquitetura NUMA entrega para o escalonador de processos do SO o controle sobre os dados e a localidade das *threads*. O SO determina a localização da *thread* no nó NUMA no primeiro uso de uma página de memória, chamado de *first touch*. As páginas de memória relacionadas às *threads* podem ser movidas durante a execução entre os nós NUMA, assim como as próprias *threads*, por questões de desempenho.

De forma geral, o escalonador de processos do SO tenta posicionar as *threads* em nós remotos – nós diferentes daquele em que foi mapeado a memória – para equilibrar a carga dos processadores, o que leva a um não compartilhamento dos dados já presentes na memória *cache* local. Em nosso caso, o mapeamento acontece durante o processamento paralelo de consultas com muitas *threads* espalhadas por todos os nós. Durante o mapeamento das *threads* em nós remotos, durante os acessos aos dados pelas *threads* ocorre a movimentação de dados. O efeito da movimentação de dados é mais impactante quando grandes estruturas são acessadas frequentemente e compartilham muitos dados entre as *threads*. Ao final, a movimentação de dados exigida pelo processamento intra-consulta em grandes bancos de dados é ineficiente tanto no modelo relacional quanto no modelo multidimensional.

1.2 OBJETIVOS DE PESQUISA

Explorar eficientemente as arquiteturas NUMA na execução de consultas é um desafio. Portanto, a questão de pesquisa que se pretende responder nesta tese é: *Como reduzir os acessos remotos à memória de modo a evitar custos extras de latência e a contenção em controladores de memória, garantindo que o SGBD aproveite eficientemente as arquiteturas NUMA e ganhe mais desempenho?*

Faz-se necessário tornar o mapeamento de threads para execução de consultas em arquitetura NUMA o mais consciente possível. Uma saída possível é a criação de um mecanismo que auxilie o SO no controle do mapeamento de *threads* durante a execução de consultas,

considerando as características de padrão de acesso à memória, as especificidades do modelo de dados do SGBD adotado e da arquitetura NUMA. Indiretamente, tal mecanismo auxiliar tornaria o mapeamento de *threads* do SGBD ciente das características do *hardware* no qual está sendo executado, otimizando o desempenho geral do processamento de consultas.

Considerando isso o objetivo dessa tese é aumentar o grau de consciência do mapeamento de threads para a execução de consultas de SGBDs na arquitetura NUMA, visando otimizar o desempenho e melhor aproveitamento das características do *hardware*. Esse objetivo tem aspectos específicos que decorrem da avaliação da execução de consulta em diferentes modelos de dados:

- Levantar o estado da arte e analisar estratégias descritas na literatura de mapeamento de *thread* para verificar seu impacto para nos SGBDs multidimensionais em execução em uma arquitetura NUMA. Assim como uma análise exaustiva de todas as possíveis combinações de mapeamento de *thread* em uma determinada arquitetura NUMA.

- Analisar o impacto da arquitetura NUMA no processamento de consultas em modelos de dados diferentes como SGBDs relacionais acesso à SGBDs multidimensionais.

- Analisar e prover um mecanismo que disponibilize gradualmente os núcleos de processamento para o SO que mapeará as *threads* para um subconjunto de núcleos para cada carga de trabalho específica, baseado nos estados de desempenho do SGBD relacional.

- Apresentar uma definição de subconjunto ótimo de núcleos baseado em limiares de uso de recursos computacionais. Explorados os limiares a partir de trabalho relacionado, ajustes empíricos e implementação do método dinâmico que considera a variabilidade de uso de recursos de diferentes cargas de trabalho.

- Propor um mecanismo de mapeamento de *thread* que assuma o controle de mapeamento e fixação das *threads* tendo conhecimento das operações de consulta em execução e da posição do endereço buscado por cada *thread*.

1.3 CONTRIBUIÇÕES

Nesta tese, é avaliado o desempenho dos SGBDs em execução em uma arquitetura NUMA. Analisa-se o impacto da arquitetura NUMA no processamento de consultas tanto no modelo relacional quanto no modelo multidimensional. Além disso, apresenta-se um mecanismo de controle de mapeamento de *threads*. De acordo com as perguntas de pesquisa elencadas, esta tese apresenta as seguintes contribuições:

- Estuda-se o impacto da arquitetura NUMA na execução de consultas e mostra-se como a entrega da responsabilidade do escalonamento das *threads* de execução de consultas para o SO afeta o desempenho da execução de uma carga de trabalho analítica.

- Estuda-se o impacto de diferentes estratégias descritas na literatura de escalonamento de *threads* para o SGBDs multidimensionais executando em uma arquitetura NUMA, incluindo todos os escalonamentos possíveis em uma arquitetura NUMA.

- Apresenta-se um mecanismo baseado em Redes de Petri, que controla de forma automática o número de núcleos e nós NUMA que o SO usa para o escalonamento das *threads* do SGBD relacional.

- Apresenta-se um mecanismo de escalonamento de *threads* baseado em Equilíbrio de *Nash*, descrito na teoria dos jogos. O mecanismo analisa os padrões de acesso à memória dos operadores de um SGBDs multidimensionais para decidir em qual nó NUMA a *thread* vai ser alocada.

1.4 LISTA DE PUBLICAÇÕES

Esta tese é baseada nas seguintes publicações originais:

- **On the performance limits of thread placement for Array Databases in Non-Uniform Memory Architectures**. Simone Dominico, Marco Antonio Zanata Alves, Eduardo Cunha de Almeida; Computing Journal [sob revisão].
- **Performance Analysis of Array Database Systems in Non-Uniform Memory Architecture**. Simone Dominico, Marco Antonio Zanata Alves, Eduardo Cunha de Almeida, Jorge Augusto Meira; 29th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2021.
- **Multi-Core Allocation Model for Database Systems**. Simone Dominico, Eduardo Cunha de Almeida; Proceedings of the VLDB Ph.D. Workshop, 2018.
- **An Elastic Multi-Core Allocation Mechanism for Database Systems.** Simone Dominico, Eduardo Cunha de Almeida, Jorge Augusto Meira, Marco Antonio Zanata Alves; 34th International Conference on Data Engineering (ICDE), 2018.
- **A PetriNet Mechanism for OLAP in NUMA.** Simone Dominico, Eduardo Cunha de Almeida, Jorge Augusto Meira; 13th International Workshop on Data Management on New Hardware (DAMON@SIGMOD), 2017.

1.5 ESTRUTURA DO DOCUMENTO

O conteúdo deste trabalho está organizado em outros seis capítulos além deste introdutório. No Capítulo 2, são apresentadas as características das arquiteturas paralelas, gerenciamento dos processos/*threads* pelo SO, modelos de SGBD e o paralelismo no processamento de consulta. O Capítulo 3 apresenta as análises das abordagens existentes que tratam o problema de movimentação de dados em arquiteturas NUMA ligadas ao contexto desta proposta. No Capítulo 4, está descrito um mecanismo dinâmico de alocação de núcleos para os SGBD relacional. Já o Capítulo 5 apresenta um estudo do impacto da arquitetura NUMA em SGBDs multidimensionais, enquanto o Capítulo 6 apresenta o mecanismo de mapeamento de *thread* para os SGBDs multidimensionais. Para finalizar, o Capítulo 7 discute as principais conclusões desta tese e traz propostas de trabalhos futuros.

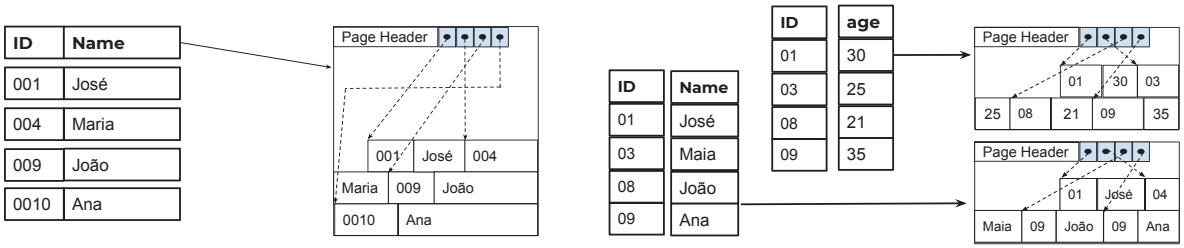
2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os modelos de SGBD, as novas arquiteturas de *hardware* e o escalonador de processos do SO considerados nesta tese. Para os SGBDs relacionais, são descritas as características dos modelos de armazenamento e as principais etapas do processamento de dados. Da mesma forma, o modelo de matriz multidimensional e o paralelismo no processamento de consulta são apresentados. Além disso, são descritas as características das novas arquiteturas de *hardware* e suas hierarquias de memória. Ainda é apresentada uma visão geral do gerenciamento dos processos/*threads* feito pelo SO. Por fim, o capítulo traz uma discussão sobre a relação dos SGBDs e as arquiteturas NUMA.

2.1 SISTEMAS GERENCIADORES DE BANCO DE DADOS (SGBDS)

Segundo [Elmasri et al. 2005] um SGBD é formado por uma coleção de programas que permite a criação e manutenção do banco de dados. Os SGBDs apresentam soluções eficientes para armazenamento e processamento de dados em diferentes modelos. Os modelos de SGBDs representam uma estrutura lógica de armazenamento de dados. Esta proposta concentra-se em dois modelos de dados: o modelo de dados relacional e o modelo de matrizes multidimensionais.

2.1.1 Modelo de Dados Relacional



(a) Orientado a linhas.

(b) Orientado a colunas.

Figura 2.1: Modelos de Armazenamento.

No modelo relacional, os dados são representados como uma coleção de relações. Uma relação é representada por uma tabela em que as linhas são um conjunto de tuplas e as colunas são um conjunto de atributos. Um atributo descreve o significado da entrada de dados na coluna. Os SGBDs relacionais possuem diferentes esquemas de armazenamento: orientado a linhas, orientado a colunas ou híbrido. O modelo híbrido particiona as relações com base no tipo de operação em execução, podendo ser orientado a linhas ou a colunas.

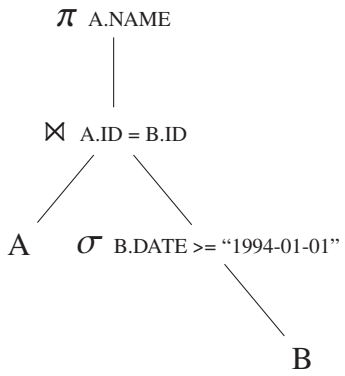
O armazenamento orientado a linhas é o esquema de armazenamento mais comumente encontrado nos sistemas comerciais. Os dados são armazenados tupla por tupla de forma contígua, como se pode ver na Figura 2.1(a). O modelo orientado a linhas é ideal para adicionar/alterar os dados (transacionais) e para recuperar todos os atributos (consultas pontuais). Contudo, na busca de um subconjunto de dados, esse modelo pode ler dados desnecessários – como atributos que não fazem parte da consulta, mas que mesmo assim são movidos até a memória cache da CPU –, tornando o processamento de consultas menos eficiente e portanto mais lento.

Já no modelo orientado a colunas, os dados são armazenados por atributo de forma contígua, como apresentado na Figura 2.1(b). Esse modelo possui um bom desempenho em

consultas analíticas que buscam um subconjunto de atributos ou consultas de agrupamentos. Somente os atributos necessários à consulta são lidos. Entretanto, para adicionar/alterar dados, são necessários vários acessos, devido à separação dos atributos em colunas. Dessa forma, o processamento de transações pode ser menos eficiente nesse esquema de armazenamento.

```
1 SELECT A.NAME
2 FROM A, B
3 WHERE A.ID = B.ID
4 AND B.DATE >= "1994-01-01"
```

(a) Consulta em Linguagem SQL.



(b) Plano de Consulta Lógico.

Figura 2.2: Consulta em linguagem declarativa (SQL) e Plano de Consulta Lógico.

O processamento de consulta é responsável por extrair informações de um banco de dados. Quando uma consulta é iniciada, usualmente escrita em linguagem declarativa SQL – exemplificada na Figura 2.2 (a) –, ela é traduzida e refinada por meio de uma série de etapas, representadas na Figura 2.3. A consulta precisa ser traduzida para a sua forma interna de entendimento para facilitar sua otimização e execução.

Durante o processo de tradução uma consulta SQL é traduzida para a álgebra relacional, gerando uma árvore de consulta algébrica (canônica) equivalente à consulta. A próxima etapa é otimizar a árvore de consulta como mostra a Figura 2.3, gerando distintas árvores de consulta otimizadas usando diferentes heurísticas. Por fim, define o plano de consulta otimizado para a execução.

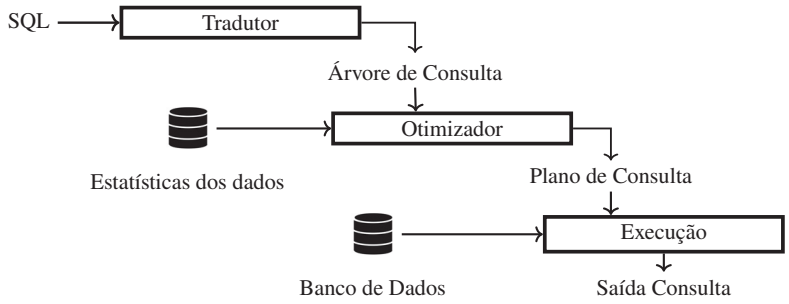


Figura 2.3: Etapas do processamento de consultas pelo SGBD relacional.

Os SGBDs relacionais utilizam a álgebra relacional para definir o plano lógico de execução de uma consulta, que pode ser visualizado como uma árvore com as operações de execução da consulta, como mostrado na Figura 2.2 (b). Na sequência, o plano lógico é otimizado e traduzido para sua representação física, com funções e rotinas para o processamento. O processamento pode utilizar diferentes algoritmos para cada operação. Cada operação é executada utilizando algoritmos específicos que são otimizados em uma das etapas do processamento de uma consulta. Os modelos de processamento de consulta podem ser materialização [Abadi et al. 2007], *pipeline* [Graefe 1990] ou vetorização [Boncz et al. 2005].

Modelo de processamento Materialização: a materialização é o processo de recuperação das informações armazenadas em forma de tuplas. A materialização pode ser antecipada, projetando as tuplas no início da consulta, ou materialização tardia, esperando o maior tempo possível para projetar as tuplas. Considerando o plano de consulta lógico apresentado na Figura 2.4 (b), a avaliação das operações na materialização ocorre no nível mais baixo da árvore de execução. As operações mais baixas são as relações do banco de dados (*A* e *B*), como visto na Figura 2.4, as operações ❶ e ❷. Então, a operação a ser avaliada é a seleção de *B* (B.DATE >= "1994-01-01") – operação ❸. A seleção é avaliada e o seu resultado é armazenado em uma relação temporária, que será utilizada como entrada no próximo operador. Como o exemplo apresentado é uma junção (Figura 2.4, operação ❹), são utilizadas a relação temporária e a relação do banco de dados *A*. A saída da junção é uma nova relação temporária. São criadas relações temporárias até a projeção, o nó raiz no plano de consulta (Figura 2.4, operação ❺).

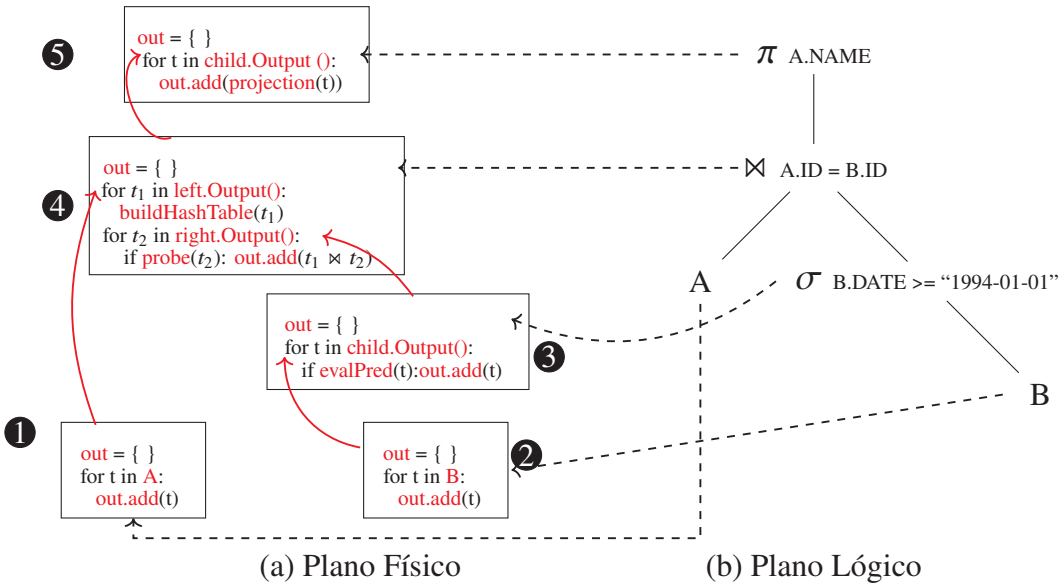


Figura 2.4: Execução de uma consulta no modelo de processamento materialização [Pavlo 2017].

Modelo de processamento *Pipeline*: no modelo *pipeline*, o resultado de uma operação no plano de consulta é transmitido para a próxima operação por meio de um *pipeline* de operações. Considerando o exemplo do plano lógico de consulta na Figura 2.5 (b), todas as operações podem ser executadas em um *pipeline*. Os resultados da seleção em B passam diretamente para a junção assim que são gerados. Os resultados de cada operação não estão disponíveis de uma só vez. A demanda em que as tuplas são solicitadas pelo operador no topo do *pipeline* – representado em ❶ na Figura 2.5 (a) – orienta a execução do *pipeline*. As tuplas são produzidas conforme a demanda pela operação seguinte, ilustrada em ❷, que avalia as tuplas fornecidas por ❸ e ❹. A operação ❹ recebe as tuplas de ❺ [Silberschatz et al. 2010].

Modelo de processamento Vetorização: o modelo de processamento de consulta vetorização suprime alguns problemas encontrados no modelo de *pipeline*. O *pipeline* transfere os dados em bloco de tuplas, sendo que cada função do banco de dados é chamada várias vezes durante a execução de uma consulta. O modelo de processamento vetorizado processa as tuplas em vetores. Normalmente, os vetores possuem tamanhos fixos para se encaixarem na memória *cache*. Sendo assim, as funções processam conjuntos de dados em fragmentos de bloco de *cache*, tornando o modelo *pipeline* eficiente [Boncz et al. 2005]. A Figura 2.6 (a) ilustra o processamento vetorizado em que são processados vetores de tuplas entre as operações representadas por ❶,

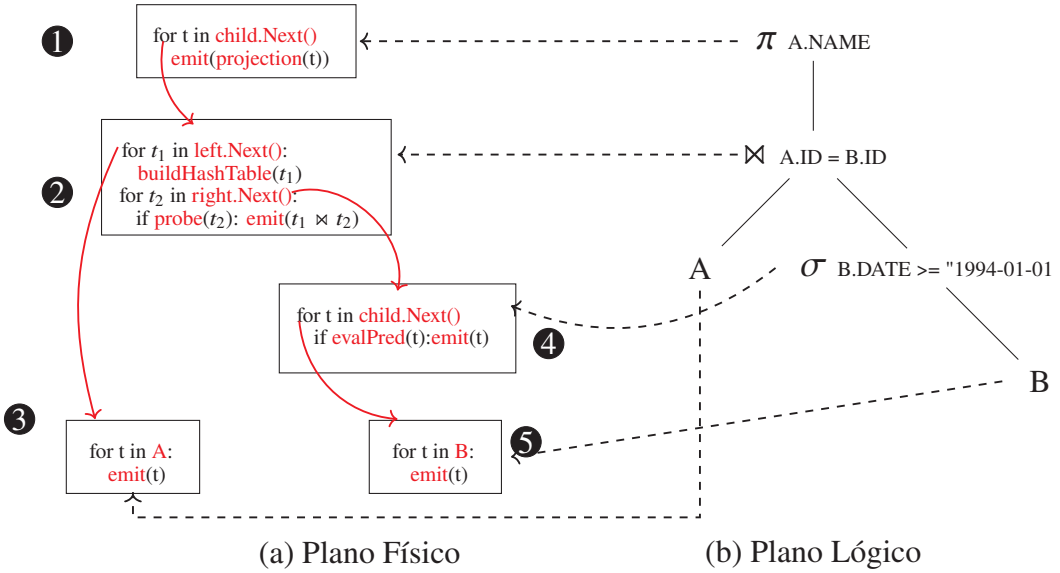


Figura 2.5: Execução de uma consulta no modelo de processamento *pipeline* [Pavlo 2017].

❷, ❸, ❹ e ❺. As funções das operações continuam solicitando os dados, assim como ocorre no *pipeline*, porém agora recebem vetores de tuplas.

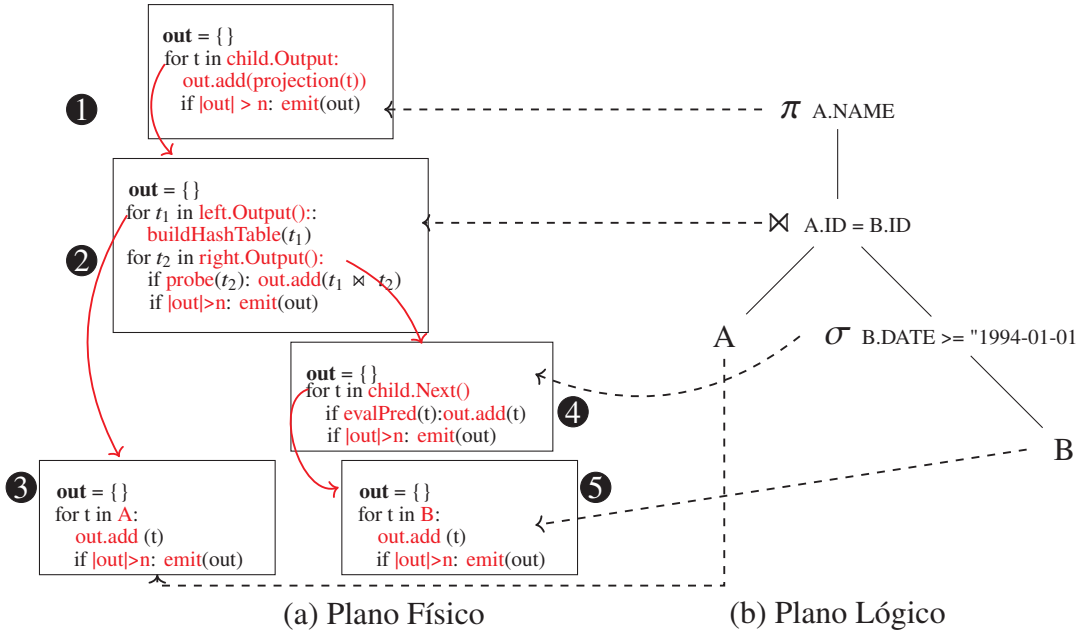


Figura 2.6: Execução de uma consulta no modelo de processamento vetorizado.

2.1.2 Modelo de Dados Multidimensional

No modelo de multidimensional, os dados são representados por matrizes multidimensionais. O modelo de dados de matriz representa dados usando *n* dimensões nomeadas e cada uma delas tem índices contíguos. As múltiplas dimensões simplificam o acesso e a análise aos dados através de diferentes visualizações. Nesse modelo, cada célula pertencente à matriz contém atributos com o mesmo tipo de dados. A Figura 2.7 ilustra o modelo de dados de matriz com três dimensões: latitude, longitude e ano. Os valores são acessíveis por um conjunto de índices. Os

usuários podem analisar rapidamente, no exemplo da Figura 2.7, mudanças na temperatura ao longo dos anos.

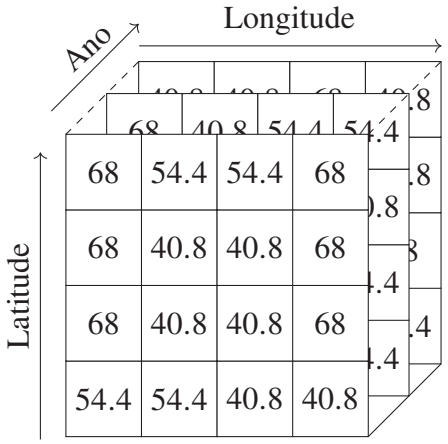


Figura 2.7: Modelo de uma matriz multidimensional: temperatura em muitas latitudes e longitudes ao longo dos anos.

Formalmente, um modelo de matriz é definido como uma tripla $A = \{S_A, D_A, M_A\}$, em que S_A é uma forma, D_A é o domínio e M_A é um mapeamento [Marathe e Salem 1999]. Uma forma é um vetor infinito de inteiros não-negativos. A forma é utilizada para definir o comprimento de A em cada número infinito de dimensões. Sendo assim, o comprimento de A na dimensão i é representado por $X[i]$. Uma célula é considerada um vetor infinito de inteiros não-negativos. Uma célula $\overline{x}$ está em uma matriz A se $\overline{x}[i] \leq A[i]\ \forall i \geq 0$. O tamanho total da matriz A é o número de células em A. O domínio D_A é um conjunto de valores não-vazio. O mapeamento M_A mapeia as células $\overline{x}$ para um valor do domínio da matriz.

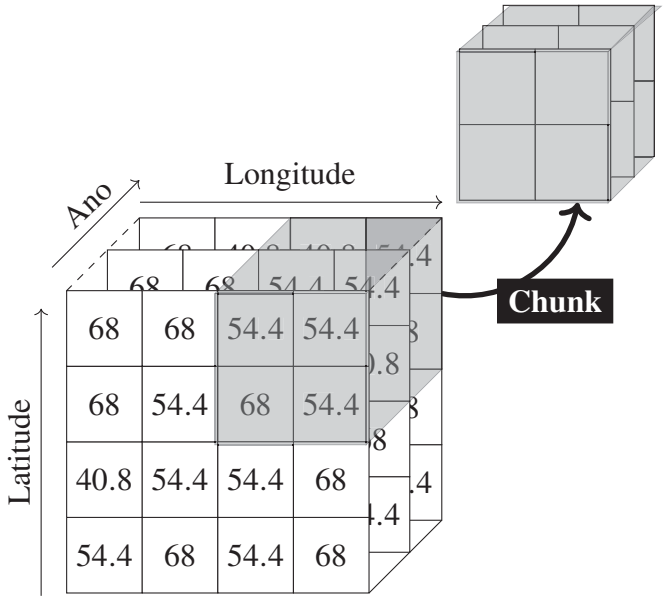


Figura 2.8: Redimensionamento de uma matriz para *chunk*.

A implementação do modelo de dados multidimensional considera o particionamento das matrizes em submatrizes, normalmente chamadas de *chunks*. A Figura 2.8 ilustra um *chunk* pertencente à matriz. Os sistemas dividem as matrizes em blocos segundo os tipos de dados armazenados. O armazenamento de uma matriz segue o modelo linear de armazenamento de

chunk, como representado na Figura 2.9. Os primeiros SGBDs multidimensionais definiam o tamanho de um *chunk* como o tamanho de uma página de memória de 64 KB, porém atualmente os *chunks* podem ter diferentes tamanhos.

O tamanho e o formato do *chunk* dependem da densidade da matriz. Para matrizes densas, os pedaços terão o mesmo tamanho no caso do SciDB [] e no SAVIME os chunks podem ser irregulares. Por outro lado, quando os matrizes são esparsas, os pedaços podem ter tamanhos e formatos diferentes.

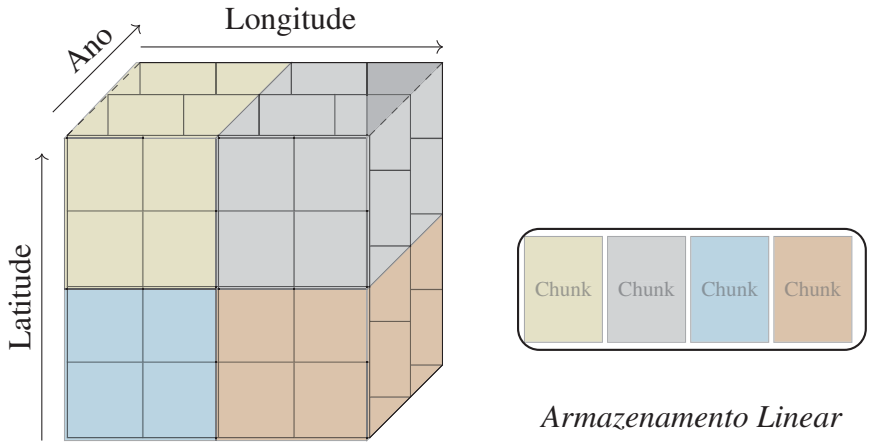


Figura 2.9: Modelo de armazenamento

As consultas em um SGBDs multidimensionais extraem dados de matrizes usando chamadas a funções aninhadas. Os *chunks* são canalizados através de operadores de consulta. Usando os índices, a estrutura de dados da matriz torna mais rápido o acesso a um conjunto de células. Além disso, se uma consulta usa frequentemente um *chunk* para acesso mais rápido, o fragmento é mantido na memória [Gerhardt et al. 2015]. Usualmente, os SGBDs multidimensionaiss utilizam uma linguagem de consulta declarativa (*Array Query Language* (AQL)) e funcional (*Array Functional Language* (AFL)). A linguagem de consulta declarativa acompanha a gramática do SQL do modelo relacional. Na fase de preparação da consulta, a AQL é transformada na AFL, destinada para realizar operações na matriz. A linguagem de consultas das matrizes permite uma flexibilidade na formulação e na otimização interna das consultas.

O modelo de processamento de consultas dos SGBDs multidimensionais é semelhante ao modelo dos SGBDs relacionais: os *chunks* são processados usando o modelo *pipeline* ou materialização. Na execução de um operador de consulta, o operador recebe uma matriz multidimensional de entrada e tem como saída uma nova matriz multidimensional com os dados selecionados. Com o conceito de modelo de dados multidimensional, uma ampla gama de SGBDs surgiu, como RasdMan [Baumann et al. 1997], ArrayStore [Soroush et al. 2011], SciDB [Brown 2010], SciQL [Zhang et al. 2013] e SAVIME [Lustosa et al. 2017].

2.1.3 Processamento de consulta paralela em SGBD relacional vs. multidimensional

A principal diferença entre os SGBDs relacional e multidimensional é o modelo de dados. As operações realizadas em um modelo de matriz multidimensional são mais complexas e estão normalmente ligadas a alto uso de CPU. Entretanto, o modelo de paralelismo pode seguir os tradicionais modelos de processamento *pipeline* ou materialização, presentes no modelo relacional, necessárias algumas modificações para a execução de operações complexas [Hahn et al. 2003].

Há um número considerável de pesquisas sobre paralelismo em SGBDs relacionais com foco em apresentar diferentes métodos de paralelismo, sendo tais métodos com arquiteturas paralelas (*shared-nothing*, *shared-disc*, *shared-everything*), intra-consulta e inter-consulta. Além disso, diferentes estratégias de execução são estudadas, como balanceamento de carga e distribuição dos dados. Algumas dessas estratégias podem ser aplicadas em SGBDs multidimensional, porém nem todas atendem à demanda desse modelo de dados.

2.1.3.1 Paralelismo de consultas no Modelo Relacional

No processamento de uma consulta pelo SGBD relacional, o paralelismo é utilizado para melhorar o tempo de execução de tarefas e o escalonamento das cargas de trabalho [Silberschatz et al. 2010]. Ao realizar operações de consulta em paralelo, é possível aumentar a quantidade de tarefas executadas pelo SGBD relacional (vazão inter-consulta) e melhorar o tempo de resposta (intra-consulta) [Silberschatz et al. 2006].

O paralelismo inter-consulta é a capacidade de executar várias consultas simultaneamente, com objetivo de aumentar a vazão. Nele, cada consulta pode ser executada por um processador de forma independente. Entretanto, o paralelismo inter-consulta não utiliza toda a capacidade de processamento da arquitetura multi-núcleo. A quantidade de processadores pode ser maior que o número de consultas em execução e algumas instruções executadas em determinado processador podem ter maior custo de processamento do que outras.

No paralelismo intra-consulta, o objetivo é minimizar o tempo de execução de uma consulta: uma única consulta é executada por vários processadores paralelamente de forma cooperativa. O paralelismo intra-consulta pode ser subdividido em duas categorias: paralelismo inter-operação e intra-operação. O paralelismo inter-operação refere-se à execução de várias operações de uma consulta em paralelo. O paralelismo intra-operação segue a premissa de que uma consulta é dividida em operações únicas, executadas em paralelo pelos processadores em subconjuntos separados de dados.

No paralelismo inter-operação, o ideal é que o algoritmo de alocação das operações coloque em um mesmo processador as operações que compartilham resultados para evitar comunicação entre os processadores. Nesse modelo, existem duas formas de paralelismo entre consulta: independente e *pipeline*. No paralelismo independente, as operações podem não depender de outra operação e serem executadas em paralelo. No paralelismo *pipeline*, as operações consomem dados de outras operações e podem ser executadas em paralelo quando recebem alguns dados.

No paralelismo intra-operação, as operações mais complexas são executadas por todos os processadores paralelamente. Essas operações são divididas em sub-operações que são executadas em partições das relações. O particionamento das relações pode utilizar diferentes critérios, como *hashing* e *range*. No particionamento de *hashing*, uma função de *hash* é aplicada em sobre um atributo chave, distribuindo as tuplas entre as partições. No particionamento de *range*, as tuplas contíguas são mapeadas para uma mesma partição [Özsu e Valduriez 1996].

2.1.3.2 Paralelismo de consultas no Modelo Multidimensional

Assim como ocorre no modelo relacional, o modelo de matriz multidimensional busca com o paralelismo aumentar o desempenho das consultas. Devido à estrutura dos dados, à quantidade de dados avaliados e à complexidade das consultas, o paralelismo de dados torna-se mais atraente para esses SGBDs. Dessa forma, os *chunks* de um modelo de matriz multidimensional podem ser direcionados para as *threads* de forma dinâmica, durante

a execução. Diferentemente do modelo relacional, que escolhe a estratégia de distribuição de dados antecipadamente, os *chunks* são solicitados sob demanda no modelo multidimensional.

Como citado, o modelo de processamento adotado por um SGBD de matriz multidimensional pode ser o *pipeline* ou materialização, assim como no modelo relacional. Entretanto, a transferência de resultados intermediários para o modelo de processamento multidimensional é mais cara devido à complexidade dos dados. A divisão de uma matriz em *chunks* facilita o processamento dos dados em paralelo, sendo que um conjunto de *threads* de uma determinada operação trabalha em um *chunk*.

Assim como no modelo relacional, o modelo de matriz multidimensional também pode apresentar paralelismo intra-consulta e inter-consulta. No paralelismo inter-consulta, as consultas são distribuídas em um *pool* de processos e agendadas para a execução. No paralelismo intra-consulta, operações processam os *chunks* com grupos de *threads*, designados para execução em nós de computação ou entre os núcleos disponíveis na arquitetura.

Atualmente, os processadores possuem múltiplos núcleos e diferentes compartilhamentos na hierarquia de memória que influenciam no processamento de consultas. Portanto, pode ser esperado um aumento de desempenho, considerando que uma execução paralela pode ser beneficiada pelos múltiplos núcleos de processamento. Entretanto, o SGBD utilizado deve ser consciente do *hardware* adjacente.

2.2 ARQUITETURAS DE PROCESSADORES MULTI-NÚCLEO

Esta seção traz uma visão geral das arquiteturas de *hardware* de processadores multi-núcleo. Essas arquiteturas compartilham a memória sendo construídas com multiprocessamento simétrico ou *Symmetric Multi-Processing* (SMP). No SMP, dois ou mais processadores compartilham a memória principal e o acesso é feito através de um único controlador de memória conectado ao barramento, como mostra a Figura 2.10 que exemplifica uma arquitetura de acesso uniforme à memória (UMA).

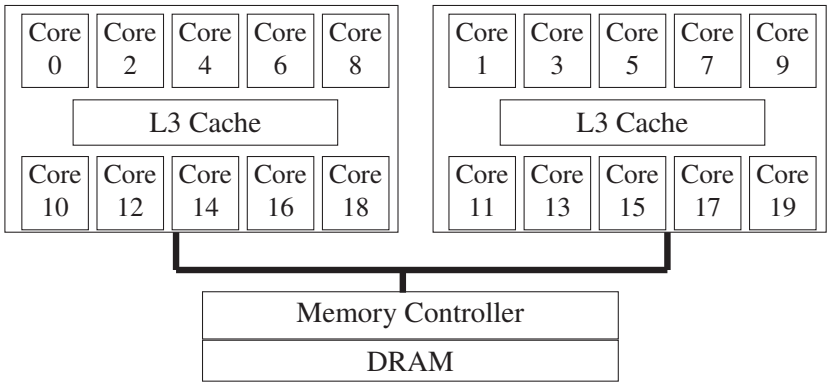
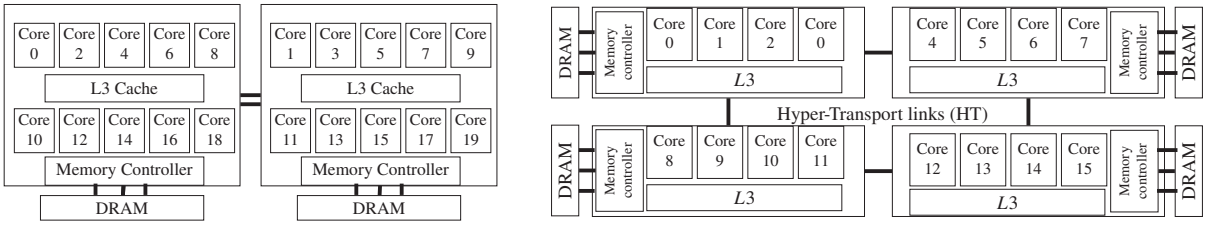


Figura 2.10: Arquitetura UMA com processadores multi-núcleo e um único controlador de memória interconectado por um único barramento.

Nas arquiteturas UMA, os processadores acessam a memória principal através do mesmo barramento. Entretanto, o compartilhamento de um único barramento de memória conduz a problemas de escalabilidade. Quanto maior o número de processadores, mais a contenção no barramento de memória é elevada e a adição de novos processadores não é mais efetiva. Para a solução desse problema, foram criadas as arquiteturas NUMA.

As arquiteturas NUMA são geralmente formadas por nós com processadores multi-núcleos, cada qual com sua própria memória local acessível a todos os nós na arquitetura, como

mostra a Figura 2.11. Dessa forma o espaço de endereçamento de memória é compartilhado e único entre os processadores do sistema. Entretanto, o acesso à memória é assimétrico, ou seja, a latência de acesso à memória é distinta em cada nó com base na localidade do dado acessado. Os acessos realizados na memória local possuem latência inferior comparados aos acessos à memória remota em outros nós. Nós remotos apresentam latência adicional com base na distância entre os nós em comunicação. A razão da diferença de latência de acesso entre o nó local e um nó remoto é chamada fator NUMA.



(a) *Intel Ultra Path Interconnect* (b) *HyperTransport Links*

Figura 2.11: Arquiteturas NUMA com processadores multi-núcleo com controladores de memória conectados com interconexões ponto-a-ponto.

A comunicação entre os nós NUMA é realizada através de uma tecnologia de banda de interconexão dos fornecedores da arquitetura [Drepper 2007]. Nas Figuras 2.11 (a) e (b), é possível ver duas tecnologias diferentes, respectivamente a *Intel Ultra Path Interconnect* (UPI)/*Intel QuickPath Interconnect* (QPI) – do fabricante de CPUs Intel – e *HyperTransport Links* (HT) – do fabricante de CPUs AMD. Essas arquiteturas NUMA apresentam diferentes topologias de acesso à memória. Os processadores podem estar conectados direta ou indiretamente à memória, variando a latência de acesso.

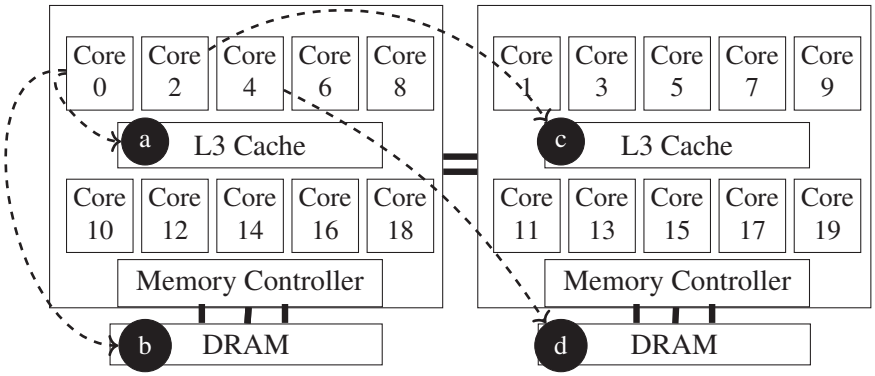


Figura 2.12: Exemplificação dos acessos à memória em uma arquitetura NUMA.

As diferenças de latência nos acessos à memória podem afetar o desempenho dos sistemas executados em uma máquina NUMA. Para evitar problemas de desempenho, os sistemas devem realizar o maior número de acessos à memória local. A Figura 2.12 exemplifica os acessos possíveis em uma arquitetura NUMA, organizados em ordem crescente de latência. Os acessos mostrados em (a) e (c) representam os acessos à *cache* local e à remota, respectivamente. São esses os acessos com o menor custo de latência. Durante uma solicitação de um endereço de memória, o primeiro acesso à memória ocorre na *cache* privada do processador ((a)). Se o endereço for encontrado, a solicitação é finalizada. Caso contrário, uma solicitação é emitida no barramento para verificar se os dados estão nas memórias *caches* de outros processadores ((c)). Se o endereço buscado não for encontrado nas *caches* dos processadores, então é solicitado à

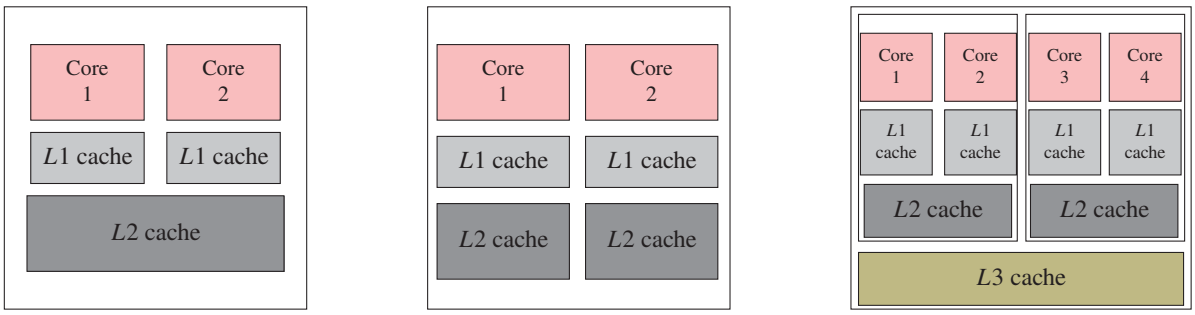
memória principal local, representada por ⓑ. O acesso local é ideal para minimizar os custos de latência impostos pela arquitetura *NUMA* na travessia da interconexão. Por fim, se o endereço não for encontrado na memória local, ele é solicitado à memória remota (ⓓ). Nesse caso, existirá uma sobrecarga de latência adicional, pois será necessário usar a interconexão de memória.

Para otimizar o desempenho de um SGBD na arquitetura NUMA, a alocação de *threads* e o posicionamento de dados devem ser realizados com reconhecimento da arquitetura NUMA. Para isso, o SGBD deve reconhecer a arquitetura NUMA e realizar um número maior de acessos à memória local, evitando gargalos de largura de banda e de interconexões [Ailamaki et al. 2017].

2.2.1 Hierarquia de Memórias *Cache*

A hierarquia de memórias *cache* nos processadores com múltiplos núcleos não sofreu muita alteração em relação à arquitetura, como mostra a Figura 2.13. A *cache* $L1$ permaneceu privada em cada núcleo, composta de *cache* de dados e instruções. A *cache* $L2$ pode ser privada em cada núcleo ou compartilhada, variando em algumas arquiteturas. E a *cache* $L3$ é compartilhada pelo processador. A latência e a capacidade das memórias *cache* são crescentes de $L1$ para $L3$. Se um dado solicitado é encontrado na memória *cache*, é nomeado de acerto de *cache* (*cache hit*). Caso contrário, ocorre um erro de *cache* (*cache miss*) e os dados são buscados nos outros níveis de *cache* até a memória principal.

A *cache* é dividida em linhas de *cache* lógico de tamanho fixo. Nas arquiteturas modernas, uma linha de *cache* é geralmente de 64 *bytes* de comprimento. Os dados de uma linha de *cache* são oriundos de endereços alinhados e coalescentes. As *caches* são baseadas no princípio da localidade, que se divide em temporal e espacial. A localidade temporal parte do pressuposto que dados acessados recentemente têm mais oportunidade de serem utilizados novamente, comparados com um dado utilizado há mais tempo. Na localidade espacial, pressupõem-se que há maior probabilidade de acessar dados e instruções próximas aos endereços acessados recentemente. Para os SGBDs, as *caches* são importantes principalmente para cargas de trabalho analíticas. Algumas comparações já foram realizadas na execução de consultas em arquiteturas modernas mostrando o tempo de execução e que o tempo de espera no processador está relacionado com um excesso de erros de *cache* [Zhang et al. 2015].



(a) *Cache* L2 compartilhadas (*Core duo* e *Core 2 Duo*)

(b) *Cache* L2 privada para cada núcleo (Pentium D)

(C) *Cache* L2 compartilhada (*Core 2 Quad*)

Figura 2.13: Diferentes arquiteturas de *cache* de processadores multi-núcleo.

A capacidade de processamento para sistemas que executam em paralelo aumenta com o crescimento do número de núcleos de um processador e com o avanço nas hierarquias de memória. Normalmente, cabe ao SO o gerenciamento das múltiplas *threads* de um sistema. A próxima seção descreve como o SO gerencia a execução de múltiplas *threads* e da memória disponível.

2.3 ARQUITETURA DO ESCALONADOR DE PROCESSOS DO SISTEMA OPERACIONAL

O SO gerencia a capacidade de computação do *hardware* fornecendo um meio para o uso adequado dos recursos [Silberschatz et al. 2001]. A estrutura do SO *Linux* tratado nesta tese é composta de *hardware*, núcleo (ou *Kernel*), biblioteca de funções padrão, *shell* e aplicações, conforme descrito na Figura 2.14. O *kernel* é responsável por controlar todas as operações que envolvem processos.

Os SOs definem políticas de gerenciamento de recursos do *hardware* para administrar conflitos quando duas aplicações tentam utilizar os mesmos recursos. Para gerenciar os recursos do *hardware*, os SOs devem possuir diferentes funcionalidades, incluindo a gerência de proteção, de arquivos, de dispositivos, de memória e de processadores. Os SOs podem executar diferentes programas em simultâneo, por exemplo, um processador de texto e um navegador *web*. A execução de um programa é nomeada de processo. Durante a execução de processos, o SO é responsável pelas chamadas de sistema que criam e gerenciam os processos [Maziero 2019].

Os primeiros SO associavam um único fluxo de execução por processo. Com o tempo, isso foi se tornando uma limitação, pois diferentes programas podem possuir vários fluxos de instruções sendo executados. Os SO atuais expandiram o conceito de processos, tornando possível a execução de múltiplos fluxos de execução, as *threads* [Silberschatz et al. 2001].

Se uma máquina que executa vários processos em simultâneo, esses processos podem ter diversas *threads* que acabam aumentando a competição pelos recursos do processador. Em geral, o *Linux* trata as *threads* como se fossem processos (*kernel threads*). Assim, o SO é responsável por decidir qual *thread* será executada. Para isso utiliza um algoritmo de escalonamento para esta decisão. Um escalonamento ótimo é um problema NP-Completo porque é um problema combinatorial que precisa satisfazer diferentes objetivos conflitantes [Ullman 1975], sendo alguns deles o tempo de resposta rápido, bom *throughput* para processos em *background*, evitar postergação indefinida, conciliar processos de alta prioridade com baixa prioridade para citar apenas alguns. As regras que o SO utiliza para gerenciar os diferentes objetivos são chamadas políticas de escalonamento [Tanenbaum 2004].

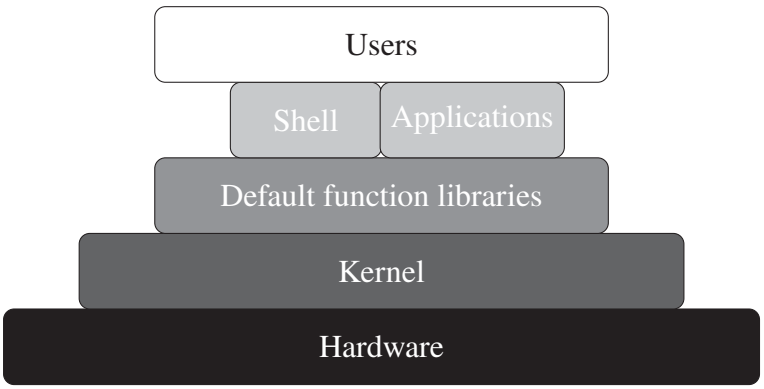


Figura 2.14: Principais componentes da arquitetura do SO *Linux*.

Os algoritmos de escalonamento de forma geral precisam atender alguns objetivos, como definir o tempo de processador para as *threads*, garantir que as políticas do sistema sejam cumpridas e manter todas as partes do sistema ocupadas. Os algoritmos de escalonamento podem ser classificados como não-preemptivos e preemptivos. Em um algoritmo não-preemptivo, a *thread* possui o controle sobre o núcleo do processador. Já em um algoritmo preemptivo, o escalonador do SO tem liberdade para movimentar as *threads* entre os núcleos ao longo da execução. No SO *Linux*, o escalonador é preemptivo e possui uma fila de prioridade dinâmica. A prioridade é modificada a partir do comportamento da *thread*.

Usualmente, as *threads* são divididas em três classes: interativas, *batch* e de tempo real. Além disso, são subdivididas em *I/O bound* ou *CPU bound*. O escalonador do SO *Linux* não efetua distinção entre *threads* interativas e de *batch*. Os escalonadores dos SOs privilegiam as *threads I/O bound* em relação às *CPU bound*, oferecendo melhor tempo de resposta às aplicações interativas [Oliveira et al. 2009].

O escalonador, além de decidir quais são *threads* serão executadas, tem a responsabilidade de decidir em qual núcleo do processador irá executar cada uma delas, mantendo o balanceamento da carga [Silberschatz et al. 2014]. O balanceamento de carga é necessário para utilizar de forma homogênea os recursos disponíveis. Entretanto, considerando arquiteturas NUMA, o problema se torna mais complexo e a diferença de latência de acesso entre os nós NUMA pode reduzir o desempenho geral da aplicação em execução [Pilla et al. 2011]. O SO normalmente possui conhecimento da arquitetura, porém não tem conhecimento do sistema em execução e busca manter o balanceamento da carga entre os nós NUMA. Para isso, muitas vezes uma *thread* pode ser migrada para um nó NUMA diferente. Na migração de uma *thread*, o SO pausa a execução atual e inicia uma nova execução em outro núcleo de processador. Durante o processamento de consulta, a migração de uma *thread* pode afetar o desempenho de um SGBD. Isso ocorre devido ao aumento de acessos remotos causado pela necessidade de migração dos dados buscados pelas *threads* que foram alocadas em outro nó NUMA [Kiefer et al. 2013].

2.3.1 Gerência e Alocação de Memória

Durante o processamento de dados, o processador acessa a memória *cache* em busca do dado necessário. Caso não encontre estes dados em *cache*, o processador buscará na memória principal e, se necessário, acessa ainda a memória virtual (disco). A memória principal é um recurso escasso. O SO precisa gerenciar a alocação de memória na execução de múltiplas *threads*. Para um gerenciamento eficiente da memória principal pelo SO, é fundamental a tradução entre endereços físicos e endereços lógicos. Na Figura 2.15, o endereço lógico é gerado pelo processador na execução de uma instrução e a tradução para endereço físico correspondente é realizado pela Unidade de Gerência de Memória (*Memory Management Unit* (MMU)). O processador pode acessar o conteúdo da memória após a tradução de endereços.

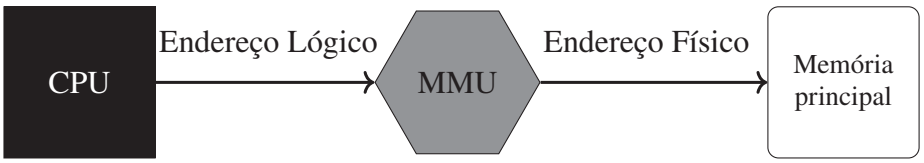


Figura 2.15: Mapeamento entre endereço lógico e endereço físico pela MMU.

Os SOs baseados em *UNIX* utilizam paginação sob demanda para decidir quais páginas são trazidas do disco para a memória. Na paginação, o espaço de endereçamento lógico do processo/*thread* é dividido em pequenos blocos com tamanhos iguais nomeados de páginas (normalmente 4 KB), assim como o espaço físico da memória é dividido em blocos denominados de *frames*. O SO gerencia os espaços livres na memória e aloca as páginas conforme os processos/*threads* vão sendo executados. A associação entre páginas e *frames* é feita por meio da tabela de páginas (*page table*).

Quando uma *thread* acessa um espaço de memória pela primeira vez e a página ainda não foi referenciada, a MMU gera uma falha de página (*page fault*), que consiste em uma interrupção de *hardware*. Durante o tratamento dessa falha de página, a página é alocada na memória utilizando uma política de alocação. No SO Linux, a política padrão de alocação de

memória é chamada de *first touch*. Nessa política cada página de memória será alocada no nodo NUMA que fizer o primeiro acesso a ela.

Considerando arquiteturas NUMA, a alocação de memória baseada em *first touch* pode afetar o desempenho da aplicação [Gaud et al. 2015]. Isso ocorre porque o padrão de acesso da aplicação pode ser modificado e todas as referências de memória se tornarem remotas. Considerando que a migração de páginas esteja habilitada, as tentativas de mover os dados para os nós com maior número de referências (migração de páginas) serão realizadas. Além disso, as *threads* vão ser atribuídas a todos os nós NUMA a partir da disponibilidade do núcleo de processamento. Portanto, as referências à memória vão ser novamente em nós remotos, adicionando o custo de latência [Diener et al. 2015].

2.4 NUMA E SGBD

Para projetar sistemas eficientes, é necessário o conhecimento do *hardware* em execução para se obter desempenho ideal. Aplicações sensíveis ao desempenho, como os SGBDs, deparam-se muitas vezes com políticas padrão, fornecidas pelo SO [Stonebraker 1981]. Considerando um *hardware* com muitos núcleos como a arquitetura NUMA, as políticas padrão do SO acabam resultando em problemas de desempenho e uso ineficiente de recursos [Giceva et al. 2016].

O SO mapeia as *threads* de processamento de consultas para todos os núcleos disponíveis entre os nós NUMA usando a política alocação [Memarzia et al. 2020; Lozi et al. 2016a,b]. No entanto, os resultados experimentais conduzidos nesta tese mostram que esse mapeamento padrão pode resultar em atividade de memória ineficiente porque os dados compartilhados podem ser acessados por *threads* dispersas que exigem grandes movimentações de dados. Além disso, dados não-compartilhados entre as *threads* podem ser alocados na mesma memória *cache* em uso por outras *threads*, aumentando os conflitos de *cache*.

Para fazer uso ideal da arquitetura NUMA, são necessários ajustes para tornar a comunicação eficiente entre os SGBDs e o SO, gerenciando as *threads* de forma inteligente [Ailamaki et al. 2017]. O ideal é mapear as *threads* para o nó NUMA em que está alocada a maior parte da memória relacionada aos dados processados, aproveitando dados alocados na hierarquia de *cache* [Giceva et al. 2013; Kiefer et al. 2013]. O capítulo 3 traz os trabalhos relacionados ao processamento de consultas de banco de dados em arquiteturas NUMA.

2.5 CONSIDERAÇÕES

Neste capítulo, foi apresentada uma visão breve e geral do paralelismo em SGBD relacional e SGBDs multidimensionais, descreveu-se como funciona o paralelismo no processamento de consultas apresentando os modelos mais utilizados. Além disso, o capítulo apresentou definições de processadores multi-núcleo, as diferentes arquiteturas e abordou as limitações que designaram a criação de multiprocessadores como a arquitetura NUMA. Também foi descrito como o SO gerencia as *threads* e decide a alocação nos núcleos.

Por fim, foram discutidos os desafios encontrados na execução de SGBDs não-conscientes da arquitetura NUMA e os possíveis problemas na execução das *threads* gerenciadas somente pelo SO. No próximo capítulo, estão elencadas as pesquisas relacionadas e os esforços para melhorar o mapeamento de *threads* em arquiteturas NUMA. Em seguida, são discutidas as soluções existentes que avaliam os efeitos da NUMA nos SGBDs modernos e durante o processamento de consultas.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta os principais trabalhos que propõem soluções para mitigar a movimentação de dados em arquiteturas NUMA. São discutidas soluções descritas por pesquisadores de arquitetura de computadores para processamento de dados e o possível impacto no processamento de consultas. Na sequência, discutem-se as soluções desenvolvidas diretamente para o processamento de consultas de banco de dados em arquiteturas NUMA. A análise apresentada foca em identificar as características de cada solução e descrever seus objetivos, evidenciando suas limitações para o embasamento da presente pesquisa.

3.1 MAPEAMENTO DE *THREADS* NO PROCESSAMENTO DE DADOS

O objetivo da arquitetura NUMA é aumentar a largura de banda disponível para acessar a memória principal. Entretanto, desde sua concepção, esse tipo de arquitetura sofre muitos problemas de desempenho durante o processamento de dados devido ao mapeamento ineficiente de *threads*. No caso particular de processamento de consultas, o SO mapeia as *threads* dos SGBDs para os núcleos disponíveis entre os nós NUMA durante a execução paralela de consultas e tal comportamento afeta o desempenho do processamento de consultas. Dados compartilhados podem ser acessados por *threads* espalhadas que requerem grandes movimentos de dados e dados não-compartilhados podem ser alocados na mesma memória *cache*, aumentando os conflitos de *cache* [Memarzia et al. 2020; Lozi et al. 2016a,b]. Discutimos à seguir os esforços que estão sendo feitos com objetivo de mitigar a movimentação de dados na arquitetura NUMA e otimizar os acessos à memória local, reduzindo a latência imposta aos acessos remotos.

O trabalho de Sergey et al. [2011] discute a necessidade de um algoritmo para reduzir a contenção em recursos compartilhados. Os autores apresentam um algoritmo de contenção de recursos compartilhados em NUMA, que migra os dados junto com as *threads*. O algoritmo utiliza técnicas de predição baseadas em informações de *miss* no último nível de *cache* (LLC – *last level cache*) para definir quais *threads* estão causando contenção de recursos. Com essa pesquisa, os autores discutem o impacto da migração de *threads* e minimizam as migrações para evitar contenção de recursos. Entretanto, os autores não tratam cargas de trabalho que compartilham dados entre as *threads*, o que pode levar a migrações de *threads* e acessos remotos que afetam negativamente o desempenho do processamento de dados. Este é o caso de cargas de trabalho do processamento inter-consulta em um SGBD, as *threads* compartilham dados durante as operações de processamento de dados. Desse modo, a migração de *threads* com o objetivo de evitar a contenção de recursos irá aumentar o número de acessos remotos e trará custos de latência adicionais.

O trabalho desenvolvido por da Cruz et al. [2012] apresenta uma abordagem de mapeamento de *threads* e dados para minimizar os efeitos da arquitetura NUMA. O mecanismo desenvolvido analisa os traços de memória com objetivo de definir padrões de comunicação entre as *threads* para criação de uma matriz de comunicação. Essa matriz é representada em um grafo, que é processado produzindo pares de *threads*, maximizando a quantidade de comunicação. Para isso, os dados são analisados de forma independente da aplicação em execução na arquitetura NUMA e o mecanismo utiliza padrões de acesso capturados por meio dos traços de memória. A correspondência entre as *threads* é representada pelo peso das arestas. Entretanto, um mapeamento estático para SGBDs pode não apresentar os melhores cenários de desempenho na arquitetura. Isso porque analisar os padrões de uma forma estática pode propiciar erros de

mapeamento no processamento de consultas, visto que os padrões de desempenho e a quantidade de dados analisada modificam-se completamente entre operações dentro de uma mesma consulta, bem como entre consultas distintas. Kepe [2019] estudaram os diferentes padrões de acesso à memória nas operações de processamento de consultas em banco de dados. O autor mostra que cada operação possui um padrão e desempenho distintos. Por exemplo, a operação de seleção apresenta acesso coalescente aos dados e baixo reuso, enquanto uma operação de junção possui acesso randômico à memória e alto reuso de dados. Os achados desse estudo reforçam, portanto, que um mapeamento estático pode não atender aos diferentes padrões de acesso à memória.

Pilla et al. [2012] apresentam uma abordagem de balanceamento de carga para máquinas NUMA usando o modelo de programação paralela *Charm++* [Zheng et al. 2017]. Utilizando *Charm++* é possível reconhecer os custos de movimentação de dados e o padrão de acesso das aplicações por meio da coleta de estatísticas em tempo de execução. A abordagem apresentada combina informações da topologia NUMA com as estatísticas coletadas usando *Charm++* e designa as tarefas intensas de CPU aos núcleos subutilizados, usando um cálculo de sobrecarga. Semelhante à pesquisa apresentada nesta tese, o objetivo dos autores é minimizar os acessos remotos na arquitetura NUMA. Os autores efetuam as avaliações de desempenho da comunicação da arquitetura NUMA antes da execução de qualquer sistema utilizando o custo de acesso em um banco de memória e o custo de latência para acessar os nós. Um ponto que difere desta pesquisa é o sistema analisado: os autores apresentam a abordagem para aplicações criadas utilizando o modelo de programação paralela *Charm++*. Nessa tese são abordadas soluções que não estão relacionadas com um modelo de programação paralela de alguma linguagem.

Um estudo sobre as políticas de mapeamento de dados para aplicações paralelas na arquitetura NUMA foi apresentado por Diener et al. [2015]. Os autores descrevem métricas importantes para análise do comportamento de acesso à memória. Com base na quantidade de acessos a uma página de memória, eles implementaram políticas de mapeamento de dados que avaliam o balanceamento de carga e o comportamento dinâmico de acesso à memória. Além disso, os autores compararam diferentes políticas de mapeamento que mostram a ineficiência da política de *first touch* padrão do Linux, como discutido na Seção 1.1 desta tese. A análise de políticas de mapeamento de dados mostra que aquelas que consideram a localidade dos dados apresentam um desempenho melhor quando comparadas às políticas que procuram o balanceamento. A alocação das *threads* durante o mapeamento de dados foi realizada de modo estático, considerando o nível de compartilhamento de páginas. Isso porque os autores focam nas políticas de alocação de páginas e não-alocação de *threads*.

O algoritmo de mapeamento dinâmico desenvolvido por Lepers et al. [2015] escolhe um subconjunto de nós NUMA para as aplicações. O objetivo do algoritmo é maximizar a largura de banda de comunicação entre processadores e entre processador e memória. O algoritmo migra as páginas de memória e as *threads*, espalhando as páginas que são compartilhadas em todos os nós NUMA. Os autores analisaram o impacto das diferentes latências de acesso de uma arquitetura NUMA e o impacto da escolha de um subconjunto de núcleos para a execução de uma aplicação. Considerando o processamento de consultas, a migração de páginas e *threads* pode ter um impacto negativo na execução de uma determinada carga de trabalho. Além disso, a alocação de páginas em todos os nós NUMA torna a utilização de memória ineficiente quando se trata de um volume maior de dados, como acontece nos SGBDs. Segundo Kiefer et al. [2013], se a quantidade de dados é maior do que a *cache* disponível, sucessivas migrações aumentam os acessos remotos.

Barrera et al. [2018] apresentam uma técnica baseada em um modelo de grafo de tarefas para minimizar os custos adicionais de latência causados por acessos remotos na arquitetura NUMA. As informações do sistema são coletadas em tempo de execução criando um grafo de

dependência de tarefas (TDG - *task dependency graph*) gerado pelo OpenMP [Committee 2013]. Utilizam-se algoritmos de particionamento de grafos para decompor o TDG em partes. As partes do TDG correspondem à quantidade de nós da arquitetura NUMA. Ao agendar uma tarefa, é explorado o tamanho das dependências – obtido por meio do TDG –, atribuindo ao nó NUMA um peso. A tarefa será designada ao nó com maior peso. Nos resultados mostrados pelos autores o movimento de dados é reduzido, porém, tal técnica é aplicada para sistemas que reconhecem as dependências do paralelismo. Nesta tese, são considerados os SGBDs que deixam para o SO o mapeamento das *threads* do processamento de consultas. Sendo assim, a técnica apresentada por Barrera et al. [2018] se aplicada ao processamento de consultas pode causar uma sobrecarga para criação de grafos de dependência em tempo de execução. Além disso, a interação entre as *threads* pode mudar ao longo da execução sendo necessária a nova organização do TDG.

Diener et al. Diener et al. [2014, 2016, 2017] propuseram um mecanismo para mapeamento de *threads* e dados nomeado de *kMAF*, que é implementado no *kernel* do Linux. Utilizando estatísticas de falhas de página, o *kMAF* projeta um perfil do padrão de acesso à memória das *threads*. É analisada a vantagem de realizar uma migração de página para outro nó NUMA. Frequentemente, é executado um algoritmo para determinar a necessidade de migração de dados. Os autores mostram ganhos de desempenho e economia de energia, porém não consideram informações específicas da aplicação antes que um determinado componente de *software* comece a coletar informações. Isso leva o mecanismo a agir quando já apresenta acessos remotos. Além desse, outros trabalhos [Dashti et al. 2013; Cruz et al. 2016a; Di Gennaro et al. 2016] também buscaram estratégias baseadas em perfil *online* coletando informações dos contadores de *hardware*.

Cruz et al. [Cruz et al. 2016a,b] apresentaram um assistente em *hardware* que usa informações relacionadas ao tempo que cada página reside na *Translation Lookaside Buffer* (TLB)[1] para melhorar o mapeamento de dados. O mecanismo fornece ao SO o melhor mapeamento baseado nos acessos à memória. Os autores apresentam uma discussão sobre o consumo de energia, mostrando que a eficiência energética aumenta em até 15.7% quando existe um bom mapeamento de dados e *threads*. Entretanto, caso aconteça uma mudança nas entradas da TLB o mecanismo pode não considerar esse padrão, pois verifica somente os erros na TLB. Além disso, para aplicar essa técnica, são necessárias modificações no *hardware* e na tabela de páginas.

Muitas pesquisas focaram diretamente no impacto do escalonamento de *threads* pelo SO. Lozi et al. [2016a,b] exploraram as deficiências do SO e analisaram as ineficiências na utilização dos recursos disponíveis em arquiteturas multi-núcleo. Os autores mostram que os escalonamentos do SO podem levar o sistema em execução a apresentar um baixo desempenho. O desempenho reduz porque, enquanto algumas *threads* estão esperando em uma fila para a execução, alguns núcleos estão ociosos, o que revela um problema com o escalonamento do SO. Além disso, os autores mostram que os escalonadores do SO tornaram-se tão complexos com a arquitetura NUMA, que acabam causando problemas no escalonamento das *threads* que estão em estado de espera. As *threads* em espera acabam paradas em filas de execução por alguns segundos, mesmo tendo núcleos ociosos. Uma das cargas de trabalho de teste utilizada pelos autores consiste em consultas analíticas. Assim como nesse trabalho correlato, nesta tese é analisado o impacto do escalonamento do SO e de diferentes técnicas para verificar o impacto no desempenho de SGBDs. Em contraponto, esta tese apresenta mecanismos que auxiliam e assumem o controle da alocação de *threads* de processamento de consulta.

Os impactos da arquitetura NUMA em diferentes sistemas continuam sendo discutidos pelas pesquisas recentes. Chiang et al. [2018] apresentam um mecanismo para melhorar o

[1]Dispositivo de *hardware* implementado a partir de uma memória associativa que mapeia endereços virtuais em endereços físicos sem passar pela tabela de páginas.

balanceamento de carga em arquiteturas NUMA. O mecanismo apresentado reconhece os acessos à memória e as políticas de alocação com objetivo de reduzir os acessos remotos. Para isso, o *kernel* do Linux é modificado e são adicionadas políticas que selecionam as *threads* para a migração. O mecanismo rastreia a quantidade de memória utilizada por cada *thread* e decide o candidato ideal para a migração. Tal mecanismo difere desta tese porque busca, por meio de modificações no *kernel* do Linux, opções para decidir a migração de *threads*. Nesta teste, argumenta-se que é necessário conhecer as características do processamento de consulta e da topologia para decidir uma alocação que atenda a uma determinada carga de trabalho.

Serpa et al. [2018] apresentam um estudo do impacto de estratégias de mapeamento de *threads* e dados na arquitetura NUMA. O foco principal do estudo é analisar o desempenho de algoritmos de aprendizado de máquina. Os autores evidenciaram as diferentes características dos algoritmos analisados e verificaram o impacto de cada estratégia utilizada. Semelhantes à pesquisa apresentada nesta tese, as estratégias de mapeamento são utilizadas para analisar o comportamento dos sistemas. Nesta tese, também apresentamos uma análise estática. Entretanto, a pesquisa desta tese traz resultados diferentes, evidenciando que as cargas de trabalho de processamento de consultas diferem de algoritmos de aprendizado de máquina.

O trabalho de Wang et al. [2016] utiliza a predição do uso largura de banda da memória para encontrar a alocação ideal de núcleos. O mecanismo considera os recursos da memória e a assimetria da arquitetura NUMA para decidir, por meio da programação linear inteira, como será a alocação de núcleos. Os fatores que influenciam a alocação ideal de núcleos incluem acessos locais, largura de banda máxima dos *links* de interconexão e contenção entre os acessos à memória local e os nós. Esta tese apresenta o conceito de número ótimo de núcleos, porém difere da pesquisa apresentada anteriormente porque são avaliados os SGBDs relacionais e os SGBDs multidimensionais e não é utilizada predição – o número ótimo de núcleos é encontrado em tempo de execução, a partir do consumo de recursos do *hardware*.

O trabalho desenvolvido por Chasparis et al. [2017, 2019] apresenta um mecanismo baseado em teoria dos jogos. Os autores analisam o uso de processamento para definir a alocação de *threads*. Ao longo da execução, analisam padrões de processamento e aprendem com o consumo de recursos para alocar as *threads* em posições próximas aos dados. Utilizando a coleta em tempo real de informações dos contadores de *hardware*, os autores analisam o desempenho de cada *thread*. Os núcleos são tratados como locais disponíveis para a alocação das *thread*. Esse trabalho é semelhante ao mecanismo proposto nesta tese para os SGBDs multidimensionais, porém nesta tese são analisados a memória e a localidade dos dados antes de decidir a alocação da *thread*. Além disso, nesta tese são observados os padrões de acesso à memória das operações durante o processamento de consulta.

Denoyelle et al. [2019] usaram o aprendizado de máquina para detectar a sensibilidade dos sistemas às políticas de alocação de *threads* e encontrar a melhor política para cada sistema. Os autores coletam informações dos contadores de *hardware* para traçar um perfil dos sistemas e determinar a melhor estratégia de mapeamento para cada necessidade. As políticas de posicionamento de *threads* estudadas são limitadas à alocação sequencial de *threads* no mesmo nó NUMA e à distribuição esparsa entre os nós. Para o posicionamento de memória, foram utilizadas duas políticas de alocação de memória: *round-robin* e *first touch*. Limitando as políticas, o estudo não abrange uma implementação para diferentes sistemas. Ademais, para um determinado conjunto de entradas, os autores consideram que as *threads* que possuem a mesma identificação em duas execuções do sistema executam cálculos equivalentes.

Popov et al. [2019] exploraram os efeitos de alocação de *threads* e páginas, grau de paralelismo e interações entre regiões de memória na arquitetura NUMA. Todas essas métricas são avaliadas para verificar os benefícios de se otimizar conjuntamente os mapeamentos de

Tabela 3.1: Análise Comparativa entre os Trabalhos Relacionados

Trabalho	***Hardware***	***Software***	**Dinâmica**	**Estática**	***Threads***	**Dados**	***Online***	***Offline***
Sergey et al. [2011]		✓	✓		✓		✓	
da Cruz et al. [2012]		✓	✓		✓	✓		✓
Pilla et al. [2012]		✓			✓			✓
Dashti et al. [2013]		✓	✓		✓	✓	✓	
Kiefer et al. [2013]		✓		✓	✓	✓	✓	
Diener et al. [2014, 2016, 2017]	✓		✓		✓	✓	✓	
Diener et al.Diener et al. [2015]		✓	✓		✓	✓	✓	
Lepers et al. Lepers et al. [2015]		✓	✓		✓	✓	✓	
Di Gennaro et al. [2016]		✓	✓		✓	✓	✓	
Cruz et al. [2016b,a]	✓		✓		✓	✓	✓	
Wang et al. [2016]		✓		✓	✓		✓	
Lozi et al. [2016a,b]		✓		✓	✓			✓
Chasparis et al. [2017, 2019]		✓	✓	✓			✓	
Barrera et al. [2018]		✓		✓	✓		✓	
Chiang et al. [2018]		✓	✓		✓	✓	✓	
Serpa et al. [2018]		✓		✓	✓	✓	✓	
Denoyelle et al. [2019]		✓		✓	✓	✓	✓	
Popov et al. [2019]		✓		✓	✓	✓	✓	
Cruz et al. [2021]	✓	✓	✓		✓	✓	✓	
Essa tese [Dominico et al. 2021]		✓	✓		✓	✓	✓	

página e *thread* para aplicativos em arquiteturas NUMA. Os autores propuseram uma abordagem de mapeamento que extrai *codelets* representativos para encontrar o perfil de acesso a páginas de memória e combinar com as outras métricas visando encontrar o mapeamento ideal para diferentes sistemas. No entanto, otimizar apenas parte da aplicação não garante a configuração ideal para toda a carga de trabalho em execução. No caso de processamento de consultas, tal abordagem precisa analisar todas as operações em execução para obter um bom desempenho devido aos diferentes padrões de acesso aos dados.

Recentemente, Cruz et al. [2021] apresentaram um mecanismo que combina o mapeamento de *threads* e dados. Para isto, o mecanismo utiliza uma implementação de memória virtual em *hardware* e *software* para identificar os padrões de acesso à memória. Através de informações de acesso a TLB o mecanismo rastreia de maneira mais precisa os padrões de acesso à memória. Para coletar informações das páginas na TLB é necessário alterar a MMU e adicionar um mecanismo armazena o número de acesso a cada entrada da TLB. O mecanismo controla a migração de páginas utilizando uma lista das páginas e do destino, assim o SO irá verificar a lista e realizar a migração das páginas.

Muitas pesquisam investigam o impacto da arquitetura NUMA em diferentes sistemas. Uma visão geral dos trabalhos relacionados nessa seção pode ser observada na Tabela 3.1. Para esta análise, são consideradas algumas características, como: se o método ou análise foi realizada com alterações no ***hardware*** e/ou ***software*** utilizado, se o mapeamento acontece de forma **estática** – somente no início da execução – ou realizado de forma **dinâmica** – modifica baseado no comportamento da carga de trabalho – e o mapeamento de ***thread*** e dados, por fim, se os dados são coletados ***online*** ou ***offline*** indicando se os dados utilizados são coletados em tempo real. Essas pesquisas apresentadas não analisam os SGBDs ou consideram os padrões de acesso à memória das *threads* utilizadas no processamento de consulta. Evidente que um mecanismo direcionado para aplicações em geral pode melhorar uma ou outra operação de consulta. Porém, devido à variabilidade de padrões de acesso à memória nas operações, definir um mecanismo

global, que contemple todos os padrões, é um procedimento complexo. A próxima Seção descreve as pesquisas que buscam melhorar o desempenho do processamento de consultas na arquitetura NUMA. Essas pesquisas apresentam uma perspectiva diferente de soluções para minimizar os custos adicionais de latência da arquitetura NUMA e focam diretamente na alocação de *threads* e dados dos SGBDs.

3.2 SGBD PARALELOS E ARQUITETURAS NUMA

Diversas pesquisas [Porobic et al. 2012; Albutiu et al. 2012; Giceva et al. 2013; Raman et al. 2013; Balkesen et al. 2013; Bellamkonda et al. 2013; Li et al. 2013; Leis et al. 2014; Porobic et al. 2014; Giceva et al. 2014; Kissinger et al. 2014; Gawade e Kersten 2015; Psaroudakis et al. 2015, 2016; Ozturk et al. 2016; Dreseler et al. 2017; Agrawal et al. 2017; Memarzia et al. 2020; Ray et al. 2020] em banco de dados apresentam esforços para tornar os SGBDs conscientes da arquitetura NUMA. Esta seção foi dividida em duas subseções, abordando pesquisas que buscam o mapeamento de *threads* e dados e pesquisas que empregam esforços em operações de consultas específicas. Além disso, considerando as pesquisas em banco de dados, é possível observar que o foco principal é o SGBD relacional. Em contraste, a pesquisa apresentada nesta tese investiga tanto o impacto da arquitetura NUMA em SGBD relacional como também seu impacto em SGBDs multidimensionais.

3.2.1 Mapeamento de *threads* e dados

Kiefer et al. [2013] mostram que o reconhecimento da topologia da arquitetura NUMA melhora o desempenho do processamento de consultas. São apresentados o impacto da arquitetura NUMA, o comportamento de acesso à memória e os efeitos da reutilização de dados em *cache*. Os autores abordam as oportunidades referentes à arquitetura NUMA e aos SGBD relacionais. São apresentadas também as características necessárias para definir a afinidade de *threads* e particionar dados para balancear a carga. Além disso, o estudo evidencia que sistemas que compartilham dados enfrentam problemas de desempenho com as diferentes latências na arquitetura NUMA devido à comunicação intensa entre as *threads*. Como nesta tese, os autores enfatizam a importância dos padrões de acesso à memória. Entretanto, Kiefer et al. [2013] consideram os padrões de acesso à memória para realizar o particionamento de dados e não a alocação de *threads*, como nesta tese.

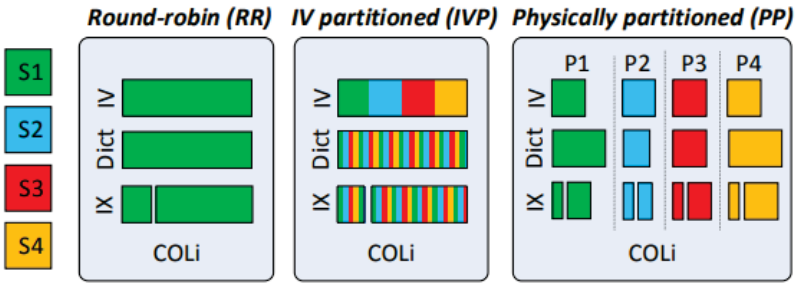


Figura 3.1: Estratégias de escalonamento de Tarefas de Psaroudakis et al. [2015].

Psaroudakis et al. [2015] apresentam estratégias de alocação de *threads* e particionamento de dados em arquiteturas NUMA para a execução de operações de varreduras concorrentes. O particionamento apresentado é dinâmico e adapta o tamanho da partição baseado na necessidade da carga de trabalho. As informações sobre os dados são armazenadas em um vetor que contém

as páginas alocadas em cada nó. Em cada nó, são alocados grupos de *threads* para a execução (ver Figura 3.1). Cada grupo de *thread* possui duas filas de prioridade divididas em tarefas. A primeira fila armazena tarefas que podem ser roubadas por outros nós. A segunda fila é exclusivamente do nó que a criou, não podendo ser roubada por outros nós. As *threads* são agrupadas de acordo com o padrão de acesso aos dados. Os resultados alcançados demonstram que o balanceamento de carga entre os nós NUMA equilibra a utilização e compartilhamento de recursos computacionais. Entretanto, os autores usam uma abordagem clássica de particionar os dados entre os nós, tratando cada nó separadamente na arquitetura NUMA. Isso pode causar uma sobrecarga de um determinado nó, sendo necessário realizar o particionamento dinâmico – que pode causar movimentação de dados excessiva.

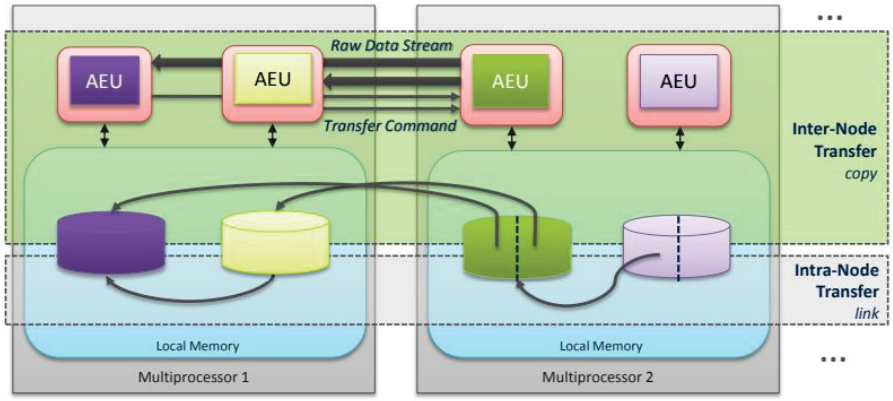


Figura 3.2: Transferência de partição realizada por ERIS entre nós NUMA [Kissinger et al. 2014].

O trabalho de Kissinger et al. [2014] descreve um mecanismo de armazenamento em memória com reconhecimento da arquitetura NUMA, nomeado de *ERIS*. O mecanismo utiliza particionamento dinâmico atribuindo a cada *thread* uma partição de dados. As *threads* são distribuídas uma para cada núcleo de processamento. *ERIS* explora a topologia NUMA e realiza o agrupamento das partições pertencentes à mesma operação para minimizar a comunicação entre os nós. As partições são transferidas entre os nós NUMA, como mostra a Figura 3.2. O *ERIS* designa uma *thread* para um núcleo de forma estática e trata a arquitetura NUMA como um sistema distribuído. Assim, o mecanismo utiliza toda a arquitetura NUMA e move grandes quantidade de dados para manter o balanceamento, o que pode causar um movimento de dados excessivo e desnecessário.

Porobic et al. [2012] exploram o impacto da atribuição das *threads* na arquitetura NUMA considerando as cargas de trabalho transacionais. A arquitetura NUMA é dividida em combinações de ilhas de *hardware* com um determinado número de núcleos. Cada ilha possui uma instância do banco de dados. A Figura 3.3 exemplifica três configurações diferentes para a criação de ilhas de *hardware*. No entanto, as ilhas não são modificadas e seu tamanho é definido estaticamente. Além disso, não é definido um tamanho ideal para uma determinada carga de trabalho. Para um bom desempenho, isso exige que o padrão de acessos aos dados não se modifique. Mas, conforme já explicado, mudanças no padrão de acesso aos dados ocorrem constantemente no processamento de consultas, dependendo da operação em execução.

Porobic et al. [2014] propõem o mecanismo *ATraPos* que gerencia o armazenamento e reconhece as diferentes latências no acesso aos dados na arquitetura NUMA. Em *ATraPos*, é adotada a seguinte estratégia para reduzir o acesso à memória remota: o particionamento é realizado dinamicamente para alocar em cada nó NUMA uma lista de transações. As listas de transações acessam somente seu nó correspondente. As *threads* são designadas para núcleos

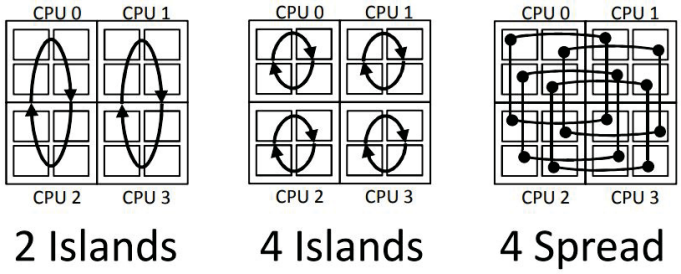


Figura 3.3: Configurações de ilhas de *hardware* propostas por Porobic et al. [2012]

de processamento específicos para garantir que as operações sejam removidas da lista de transações pela mesma *thread* que as adicionou. As partições são movidas para núcleos que estão subutilizados. Entretanto, *ATraPos* é desenvolvido para cargas de trabalho transacionais e utiliza informações das listas de transações para a alocação de *threads* em um mesmo nó NUMA.

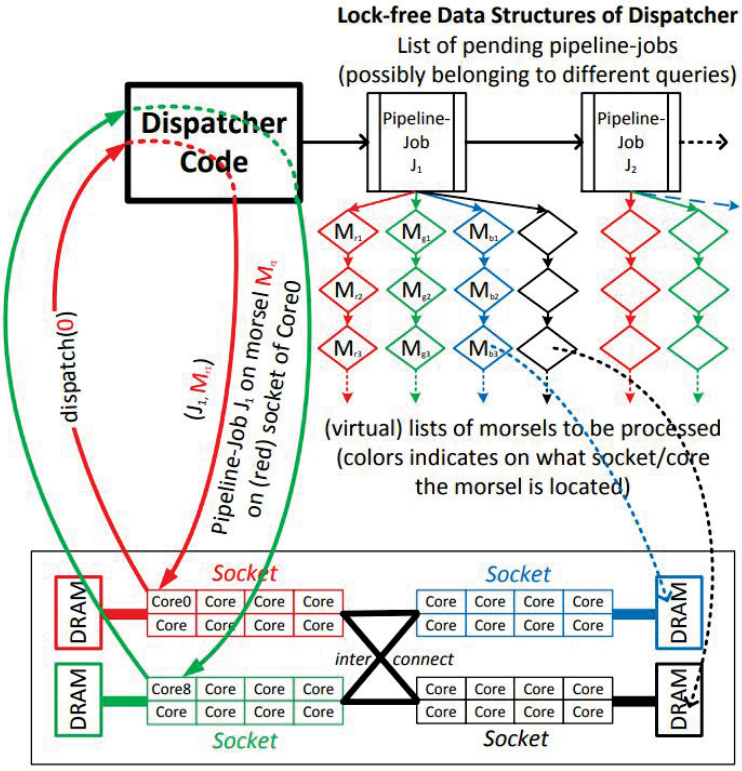


Figura 3.4: Atribuição de *threads* aos *morsels* [Leis et al. 2014].

Leis et al. [2014] trazem uma estrutura de execução paralela de consultas para o SGBD Hyper [Kemper e Neumann 2011] chamada de *morsel*. *Morsel* explora a afinidade de dados e *threads* com controle sobre o paralelismo. Os dados são divididos em pedaços (*morsels*), sendo processados em paralelo pelo mesmo *pipeline* de operador. Os *pipelines* de consultas são criados pela compilação de consulta *just-in-time* [Neumann 2011]. Cada *thread* carrega um *morsel* para o nó local em que está sendo executada (ver Figura 3.4). A afinidade de núcleo de processamento é atribuída para as *threads* para manter a localidade e estabilidade em um

mesmo nó NUMA. Com controle sobre o paralelismo e os dados, as *threads* são conscientes da localidade de dados e de cada operador. Os dados são distribuídos de maneira uniforme entre os nós NUMA, porém de forma estática. Vale ressaltar que as estruturas de *hash* utilizadas normalmente durante operações de junção e agregação são armazenadas sem o conhecimento da arquitetura NUMA. Consequentemente, podem ser armazenadas distantes das *threads* que estão utilizando esses dados e gerar acessos remotos e custos adicionais de latência [Kissinger et al. 2014].

Similar ao método de Leis et al. [2014], no SGBD DB2 BLU [Raman et al. 2013] as consultas são divididas em múltiplas chamadas de consultas, sendo cada uma executada por uma *thread*. Os dados são processados em partes por cada *thread* com tamanhos que se encaixam na memória *cache*.

O balanceamento de carga entre as *threads* é realizado por meio de "roubo de trabalho". Entretanto, o "roubo de trabalho" entre as *threads* pode incorrer em acessos remotos devido à migração de tarefas e dados [Vikranth et al. 2013].

O trabalho de Gawade e Kersten [2015] analisa a execução de planos de consulta não-conscientes da arquitetura NUMA e planos de consulta particionados que são compatíveis com NUMA. A análise compara o paralelismo do SGBD relacional MonetDB com uma implementação do MonetDB com modificações que tratam a arquitetura NUMA como uma arquitetura distribuída. O trabalho cria uma correspondência de cada nó NUMA com uma instância do MonetDB. Durante o processamento de consultas, o plano de consulta é particionado baseado no número de núcleos presentes em cada nó. A responsabilidade da alocação das *threads* é entregue para o SO, que mantém o balanceamento de carga. Além disso, o trabalho tem por objetivo avaliar o particionamento de dados na arquitetura NUMA. Isso não garante a redução dos acessos remotos: o SO tentará manter o balanceamento de carga e pode designar as *threads* para nós NUMA diferentes do local em que estão as partições correspondentes a cada uma delas.

Psaroudakis et al. [2016] mostram um protótipo de um mecanismo de particionamento adaptativo para reduzir os acessos remotos e corrigir o desequilíbrio nos nós NUMA. Esse conceito de posicionamento adaptativo pode ser observado na Figura 3.5. Utilizando dados de uso de processamento, o mecanismo detecta desequilíbrios de utilização do processador entre os nós NUMA. A partir daí, o mecanismo decide entre o roubo de tarefas ou a afinidade de dados. Essa decisão é utilizada para balancear a utilização dos recursos na arquitetura NUMA. As tarefas que são intensivas de memória não se encontram disponíveis para roubo entre os nós. Utilizando o particionamento ao longo da execução, a carga de trabalho é equilibrada entre os nós NUMA, movendo ou reparticionando as tabelas. O mecanismo utiliza particionamento dos dados, designando-os em todos os nós NUMA. As tarefas são alocadas analisando-se a intensidade de uso de processamento. A movimentação dos dados durante o particionamento pode prejudicar o desempenho devido aos custos de latência e contenção nos *links* de interconexão.

Giceva et al. [2013] propõem um sistema para melhorar a iteração entre o *hardware* e os SGBDs relacionais. O resultado é um sistema nomeado como *COD* (Figura 3.6), que combina um mecanismo de gerenciamento de banco de dados com políticas no nível de SO. No mecanismo de gerenciamento de banco de dados de *COD*, as *threads* possuem afinidade com um núcleo específico. Consequentemente, minimizam-se os erros de *cache* em níveis de *cache* privados. Considerando o SO, são mantidas informações do *hardware*, carga de trabalho do sistema e alocação de recursos. O SGBD relacional envia informações de alocação de recursos para uma estrutura que auxilia o SO a escolher a melhor alocação dos recursos. Para implementar o sistema, são necessárias modificações no SO e no SGBD relacional. O SGBD relacional precisa ter uma função de custo de previsão do impacto de alocação de recurso. Já o SO deve integrar novas regras as políticas de alocação.

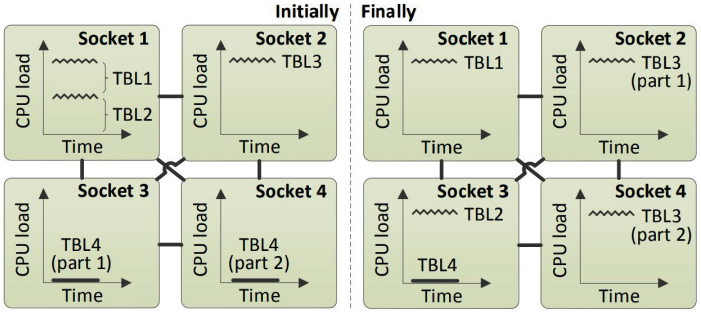


Figura 3.5: Um exemplo conceitual do posicionamento adaptativo de dados de [Psaroudakis et al. 2016].

.

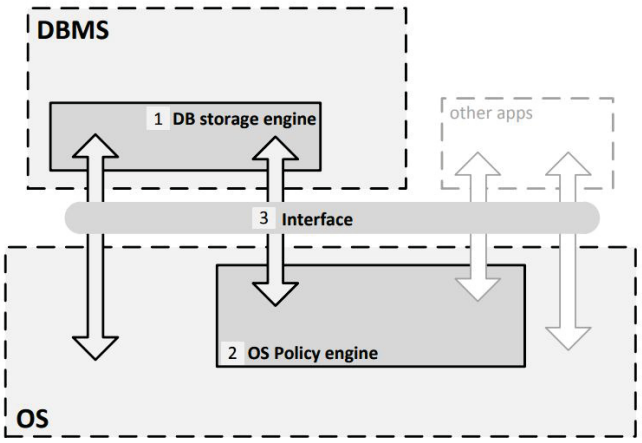


Figura 3.6: Componentes da arquitetura de COD [Giceva et al. 2013].

.

No trabalho descrito por Dreseler et al. [2017], os autores apresentam uma otimização baseada em *hardware*. As varreduras são otimizadas utilizando a API (*Application Programming Interface*) proprietária dos sistemas *SGI UltraViolet (UV) family* [SGI 2011]. Os dados são copiados de nós remotos para o nó local por meio de uma operação de cópia da API para esconder a latência do acesso remoto, como pode ser observado na Figura 3.7. Os autores apresentam duas lógicas de alocação de *threads*. A primeira aloca as *threads* de forma sequencial, preenchendo os nós NUMA. Na segunda lógica, a alocação das *threads* acontece de maneira circular (*round-robin*) entre os nós. Tais otimizações apresentadas pelos autores são designadas para máquinas específicas dos sistemas SGI UV, nas quais a latência de acesso remoto é reduzida através de um componente de *hardware* chamado de HARP[2]. Usando a Unidade de referência global (*Global Reference Unit* - GRU) do HARP, a cópia de memória entre os nós NUMA é realizada de forma assíncrona acelerando operações de memória.

Giceva et al. [2014] descrevem um algoritmo para encontrar um posicionamento dos operadores do plano de consulta. O algoritmo busca minimizar a quantidade de recursos utilizados sem afetar o desempenho e a estabilidade do sistema. Para isso, analisa-se o uso do processador e da largura de banda de memória para criar operadores denominados de compostos. O algoritmo

[2]Componente chave que conecta os processadores ao sistema interconexão NUMA.

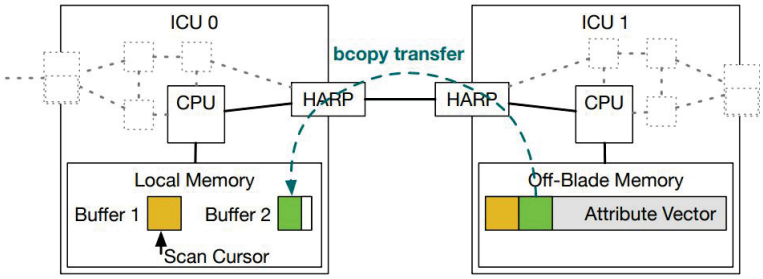


Figura 3.7: Operação de cópia utilizando a API [Dreseler et al. 2017].

.

procura minimizar a utilização de recursos, como o uso de largura de banda de interconexão e uso processamento. As entradas do algoritmo são a dependência dos operadores, o vetor de atividade de *hardware* e as informações da arquitetura NUMA – como mostra a Figura 3.8. O algoritmo detecta quais *threads* podem ser executadas em um mesmo núcleo por meio da

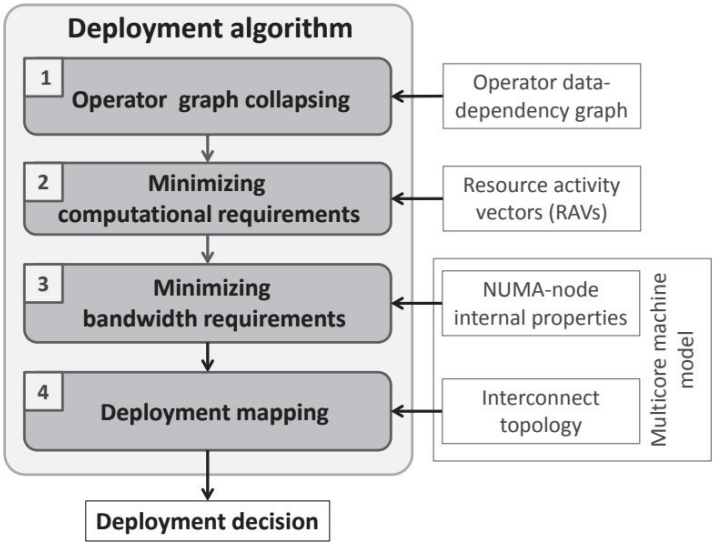


Figura 3.8: Visão geral da implementação do algoritmo de Giceva et al. [2014].

avaliação de uso do processador. Desse modo, os autores definem um número mínimo de núcleos do processador que atenda a todas as *threads*. Em seguida, definem o número de nós NUMA necessários, utilizando o mesmo algoritmo. No entanto, as avaliações são feitas *offline* para uma determinada carga de trabalho conhecida antecipadamente. Nesta tese assumimos que a carga de trabalho pode ser desconhecida, como é comum em processamento de consultas do tipo OLAP.

O trabalho de Giceva et al. [2016] expõe um *kernel* de SO para o processamento de consultas (ver Figura 3.9). O *kernel* adaptado apresenta políticas de mapeamento personalizadas, nas quais são excluídas funcionalidades que não são pertinentes para o processamento de consultas. O principal objetivo dos autores é oferecer um *kernel* com funcionalidade de mapeamento de tarefas, gerenciamento de memória e serviços pertinentes para o processamento de consultas no SGBD relacional. O protótipo é desenvolvido como extensão do SO Barrelfish [Group 2016]. A criação de um novo *kernel* otimiza as funções presentes no SO para implantação do SGBD relacional. Entretanto, as funcionalidades do SGBD relacional teriam de ser adaptadas

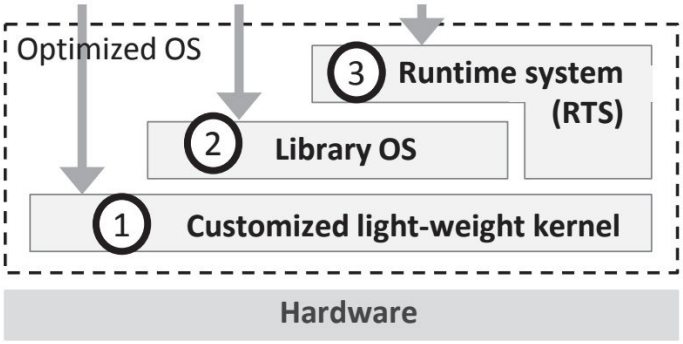


Figura 3.9: Visão geral da arquitetura otimizada do SO proposta por Giceva et al. [2016].

para a execução em um SO preparado somente para o processamento de consultas. Um SGBD específico para um determinado SO limitaria a utilização do mesmo.

O SGBD relacional *SQL Server* é consciente da arquitetura NUMA. Ele divide o plano de consulta em *branches*. Várias *threads* executam a parte do plano de consulta em um *branch* e o controle é realizado por um operador de troca [White 2013]. Além disso, o *SQL Server* apresenta a funcionalidade de *Soft-NUMA*, que pode criar nós NUMA com os processadores presentes em um único nó, ou seja, criar nós NUMA lógicos [Microsoft Docs 2018]. Para executar o *SQL Server* em um SO Linux, existe a recomendação de desabilitar o balanceamento NUMA padrão para que o *SQL Server* assuma o controle de todas as *threads* do plano de consulta.

No trabalho apresentado por Bellamkonda et al. [2013], é descrito um esquema adaptativo de distribuição de dados para o SGBD relacional *Oracle*. Nesse esquema, a chave de particionamento é aumentada para obter uma quantidade de partições semelhantes ao grau de paralelismo. Entretanto, não são claros o posicionamento de *threads* e a redução de movimentação dos dados entre os nós NUMA.

No trabalho apresentado por Agrawal et al. [2017], os autores expõem uma unidade de processamento de dados (DPU - *Data Processing Unit*) otimizada para cargas de trabalho analíticas. O sistema da DPU é composto de uma memória compartilhada por vários núcleos, com um mecanismo de movimentação de dados em que as operações de movimento e particionamento de dados são acelerados em *hardware*. A DPU possui ferramentas de *hardware* e de *software* que controlam a movimentação de dados e o particionamento. No entanto, é uma nova unidade de processamento direcionada para processamento de dados de cargas de trabalho analíticas, uma arquitetura diferente da NUMA, com otimizações específicas de movimento de dados aplicadas à DPU.

Ozturk et al. [2016] apresentam um esquema que distribui conjuntos de consultas entre os nós da arquitetura NUMA, considerando relações de afinidade. O esquema apresentado considera as hierarquias de *cache* para determinar como as *threads* são posicionadas. Para a distribuição das consultas, os autores utilizam um grafo que identifica as consultas que compartilham dados. Com objetivo de manter o balanceamento da carga e direcionar as consultas para os núcleos corretos, o esquema utiliza a programação linear inteira. Por meio do particionamento do grafo, o esquema decide a alocação das consultas observando a hierarquia de *cache*. Entretanto, o

posicionamento de consultas proposto por Ozturk et al. [2016] é realizado de forma estática. Tratando-se do processamento de consultas, o esquema deve considerar as operações executadas em cada consulta. Como dito anteriormente, ao posicionar uma consulta, os padrões de acesso à memória são modificados dependendo da operação em execução.

Recentemente, o trabalho de Memarzia et al. [2020] avalia diferentes estratégias para melhorar o desempenho de cargas de trabalho com uso intensivo de memória. As estratégias avaliadas são aplicadas em diferentes sistemas com uma perspectiva de "caixa-preta". As estratégias adotadas autores, consistem em substituir o alocador de memória e definir um posicionamento de *threads* usando um esquema de afinidade. Para isso, as estratégias adotam uma política de posicionamento de memória em que são necessárias alterações na configuração do SO. Os autores avaliaram cinco SGBDs relacionais diferentes e concluíram que algumas das estratégias usadas causam sobrecarga no desempenho do SGBD durante a execução das consultas. Além disso, mostram que os ganhos de desempenho variam baseados na posição de memória. Assim como nesta tese, os autores argumentam que SO não realiza o escalonamento ideal de *threads* para processamento de consultas. Eles concluem ainda que o escalonador do SO pode ser um gargalo para o processamento de consultas.

3.2.2 Operadores de consulta com reconhecimento da arquitetura NUMA

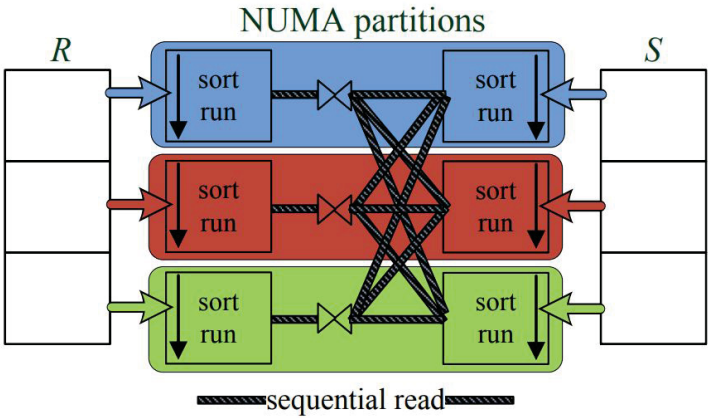


Figura 3.10: Ideia básica do algoritmo de junção apresentado em [Albutiu et al. 2012].

.

Esta Seção sumariza as pesquisas que apresentam soluções para melhorar a execução de operadores relacionais específicos na arquitetura NUMA.

Albutiu et al. [2012] propõem um algoritmo de junção (operador *join*) com reconhecimento da arquitetura NUMA para banco de dados em memória. O algoritmo realiza a pré-busca [3] para ocultar a latência de acessos remotos. Durante a junção, uma das relações é particionada em um intervalo para que a fase de mesclagem e ordenação[4] seja realizada independentemente para cada partição, enquanto a segunda relação é particionada entre todos os nós NUMA. Entretanto, a fase de mesclagem e ordenação é realizada no nó NUMA local. Já na fase junção são necessários acessos remotos, o que gera movimentação de dados e custo adicional de latência. Entretanto, os autores buscam um padrão de acesso aos dados que seja eficiente na arquitetura NUMA,

[3]Busca antecipada de instruções e dados para a memória *cache*.

[4]Junções sucessivas de relações ordenadas para criar uma relação ordenada. Baseia-se no algoritmo de dividir e conquistar.

evidenciando que observar os padrões de acesso à memória melhora o aproveitamento dos recursos disponíveis na arquitetura NUMA.

O trabalho de Li et al. [2013] propôs um algoritmo com reconhecimento de NUMA para operações de junção. O objetivo dos autores é otimizar o tráfego de interconexão. Para isso, utilizam um algoritmo baseado em coordenação de anéis que orienta o movimento de *threads* durante a recuperação dos dados. Um anel interno representa as partições de dados e um anel externo as *threads*. O algoritmo gira em sentido horário o anel interno para que todas as *threads* possam acessar todas as partições de dados correspondentes. A coordenação garante que todos os canais de memória de interconexão estejam ocupados, usando ao máximo os *links* de interconexão. O algoritmo realiza o mapeamento dos dados de forma estática e otimiza um único operador de consulta.

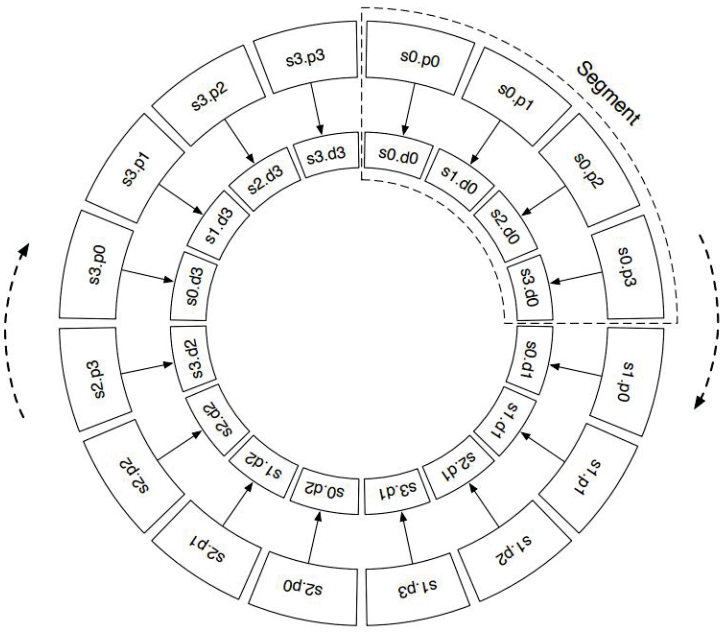


Figura 3.11: Exemplo da coordenação baseado em anel proposta por Li et al. [2013].

Já no trabalho de Balkesen et al. [2013] o foco principal são operações de junção de mesclagem. Os autores comparam os algoritmos de junção do estado da arte e analisam o desempenho com diferentes perspectivas, incluindo os efeitos da arquitetura NUMA na execução. Os autores propuseram um algoritmo que reconhece a arquitetura NUMA e otimiza a operação de junção de mesclagem. As relações da junção são distribuídas em todos os nós NUMA e as *thread* são atribuídas considerando a localidade dos dados. As partições são alocadas nos nós NUMA usando um algoritmo de classificação. Como resultado, as *threads* executam acessando somente a memória local do nó NUMA, como mostra a Figura 3.12. Entretanto, as otimizações apresentadas são designadas para uma única operação relacional, que é a junção. Os autores não consideram todas as operações do processamento da consulta e a carga de trabalho em execução na arquitetura NUMA.

Ray et al. [2020] analisam a operação de junção espacial em execução em uma arquitetura NUMA. Uma junção espacial trata-se da junção de dois conjuntos de objetos geométricos associados por um predicado espacial que identifica o relacionamento topológico entre dois objetos, como intersecção, sobreposição. Para análise da junção espacial, são exploradas diferentes políticas do SO, apontando os problemas durante a execução das junções

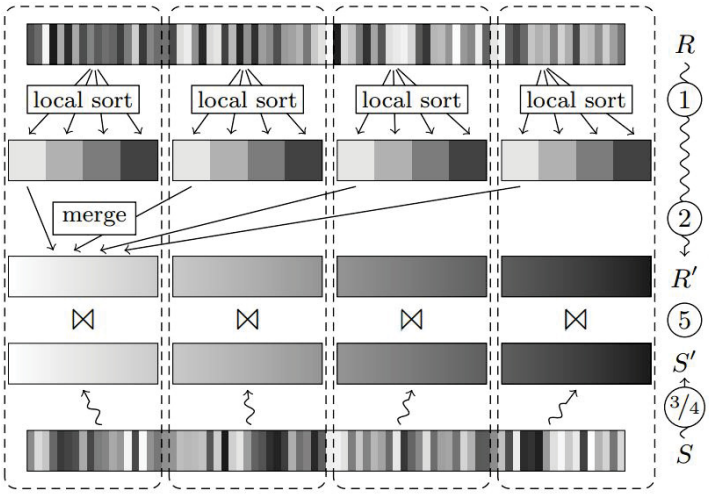


Figura 3.12: Exemplo de junção de mesclagem consciente de NUMA com mesclagem de várias vias proposto por Balkesen et al. [2013].

.

espaciais. A análise realizada mostra que, para a junção espacial, políticas de posicionamento de memória de intercalação não apresentam bons resultados. Trata-se de uma consequência do particionamento dos dados, que é realizado de forma estática e com as *threads* direcionadas para os nós NUMA. Entretanto, os autores argumentam que, para melhorar o desempenho da operação de junção espacial, é necessário um mecanismo dinâmico de posicionamento de thread e dados e particionamento de dados. Além disso, que estratégias utilizadas para a operação de junção comum não se aplicam para a junção espacial.

3.3 CONSIDERAÇÕES FINAIS

Nesta tese, são propostos dois mecanismos, um mecanismo para os SGBDs relacionais e um mecanismo para os SGBDs multidimensionais. Considerando os SGBDs relacionais, o foco é encontrar a quantidade ideal de núcleos para atender uma determinada carga de trabalho e reduzir o movimento de dados entre os nós NUMA por meio de um posicionamento implícito. Quando o texto refere-se a posicionamento implícito, significa que a afinidade das *threads* não é atribuída para um determinado núcleo. Ele apenas é disponibilizado um conjunto de núcleos elegíveis que o SO pode designar para uma determinada carga de trabalho. Para determinar o conjunto de núcleos elegíveis o mecanismo avalia a utilização de recursos como carga de CPU e através de limiares define quando um novo núcleo é ou não necessário. Nesse sentido, o mecanismo pode se utilizado em diferentes SGBDs relacionais.

Para os SGBDs multidimensionais, o foco principal é observar os padrões de acesso à memória. Isso porque para o posicionamento das *threads*, são utilizadas informações sobre erros de *cache* obtidas com os contadores de *hardware*. O mecanismo proposto nesta tese é semelhante ao trabalho descrito por Chasparis et al. [2017, 2019]. Ele utiliza a teoria dos jogos e trata cada *thread* como um tomador de decisão. Entretanto, o foco principal do mecanismo desenvolvido para SGBDs multidimensionais são os acessos à memória, ao contrário dos autores que utilizam o uso de processamento para designar as *threads*. Além disso, o mecanismo desta tese foca diretamente em SGBDs multidimensionais.

Tabela 3.2: Análise comparativa entre os trabalhos relacionados diretamente com SGBDs

Trabalho	**Dinâmico**	**Estático**	***Threads***	**Dados**	**Matriz**	**Relacional**	**Analítica**	**Transacional**
Porobic et al. [2012]		✓	✓	✓		✓		✓
Albutiu et al. [2012]		✓	✓	✓		✓	✓	
Giceva et al. [2013]		✓	✓			✓	✓	
Raman et al. [2013]		✓	✓	✓		✓	✓	
Balkesen et al. [2013]		✓	✓	✓		✓	✓	
Bellamkonda et al. [2013]	✓		✓			✓	✓	
Li et al. [2013]	✓		✓	✓		✓	✓	
Leis et al. [2014]		✓	✓	✓		✓	✓	
Porobic et al. [2014]	✓		✓	✓		✓		✓
Giceva et al. [2014]		✓	✓			✓	✓	
Kissinger et al. [2014]	✓		✓	✓		✓	✓	✓
Gawade e Kersten [2015]		✓	✓	✓		✓	✓	
Psaroudakis et al. [2015]	✓		✓	✓		✓	✓	
Psaroudakis et al. [2016]	✓		✓	✓		✓	✓	
Ozturk et al. [2016]		✓	✓	✓		✓	✓	
Dreseler et al. [2017]	✓		✓			✓	✓	
Agrawal et al. [2017]		✓	✓			✓	✓	
Memarzia et al. [2020]		✓	✓	✓		✓	✓	
Ray et al. [2020]		✓	✓	✓		✓	✓	
Essa tese [Dominico et al. 2021]	✓		✓	✓	✓	✓	✓	

Uma visão geral dos trabalhos relacionados pode ser observada na Tabela 3.2. Para esta análise, são consideradas algumas características, como: **i)** se o método utilizado é **estático ou dinâmico**; **ii)** o posicionamento de ***thread* e dados**; **iii)** o tipo da carga de trabalho **analítica ou transacional**; **iv)** se a abordagem é ***estática* ou dinâmica** e, por fim; **v)** se é aplicada a SGBD **relacional ou multidimensional**.

Em todos os trabalhos discutidos, o objetivo é analisar e minimizar os custos adicionais de latência na arquitetura NUMA. No contexto de execução de consultas, o custo adicional de latência é causado por acessos remotos. A partir dos trabalhos relacionados, neste capítulo é possível constatar diferentes estratégias desenvolvidas para melhorar o desempenho dos SGBD relacional em arquiteturas NUMA. São utilizados algoritmos de posicionamento de *threads* e dados para atingir esse objetivo. Em contrapartida, constata-se que nenhuma análise considera, além do SGBD relacional, os SGBDs multidimensionais. Além disso, mesmo considerando SGBD relacional, poucas análises propuseram um número ideal de núcleos para o processamento de consultas. Ainda é possível observar que diferentes soluções para SGBD relacional utilizam o particionamento de dados priorizando o balanceamento de carga.

Por fim, ressaltamos que nosso trabalho se beneficia do conhecimento que o SO faz o mapeamento *first-touch*. Dessa forma, quando alocamos as *threads*, estamos de fato decidindo o mapeamento de *threads* e dados, de uma só vez. Ou seja, enquanto diversos correlatos fazem apenas o mapeamento de dados, nós focamos no mapeamento de *threads* levando de forma indireta ao mapeamento de páginas.

4 UM MODELO ABSTRATO DE ALOCAÇÃO MULTI-NÚCLEO PARA BANCO DE DADOS RELACIONAL

Em sistemas modernos multiprocessados com arquitetura NUMA, formada por vários nós multi-núcleos, a hierarquia da memória torna-se mais complexa: composta por vários níveis de *cache* e diferentes esquemas de compartilhamento de memória entre os núcleos. Nesse contexto, encontrar afinidade de dados para aplicativos *multithread* é um problema altamente relevante. Considerando aplicações de processamento de dados, o mapeamento correto de *threads* e dados sobre os nós NUMA reduz a movimentação de dados, o que reduz também o número de conflitos de *cache* entre as *threads* e, consequentemente, leva a melhorias no desempenho final.

Nesse contexto, quando os SGBDs relacionais executam cargas de trabalho de processamento analítico *online* (OLAP), o paralelismo é encapsulado dos operadores isso significa que os operadores são mantidos inconscientes do paralelismo. Normalmente, o paralelismo é definido no plano de consulta no momento do planejamento e o trabalho é atribuído às *threads* estaticamente [Leis et al. 2014], [Hellerstein et al. 2007]. Nesse modelo, o SO é responsável por mapear as *threads* do SGBD relacional para o maior número possível de processadores e núcleos [Psaroudakis et al. 2016]. Isso significa que o SO tradicionalmente aloca uma *thread* por núcleo dispersamente entre todos os nós NUMA disponíveis. No entanto, se várias *threads* que compartilham o mesmo bloco de dados são mapeadas distantes entre si (por exemplo, em nós NUMA diferentes), elas podem causar uma grande movimentação de dados, aumentando o tráfego entre nós e o número de erros de *cache*. Por outro lado, se as *threads* que atuam sobre blocos privados de dados forem mapeadas nas proximidades (por exemplo, no mesmo nó NUMA), elas podem causar um grande número de conflitos de *cache*.

Este capítulo apresenta um mecanismo de alocação de núcleos de processamento para oferecer suporte à alocação de *threads* e de dados em nós NUMA. A hipótese é que é possível mitigar a movimentação de dados entre os nós NUMA se o SO mapear as *threads* apenas para um subconjunto eficiente de núcleos de processamento para cada carga de trabalho específica com base nos estados de desempenho do banco de dados. Assim, o mecanismo de alocação de núcleo analisa sistematicamente o uso de recursos de *hardware* pelas *threads* em execução para configurar o estado atual de desempenho do SGBD relacional. Em seguida, o mecanismo decide se os núcleos precisam ser alocados ou liberados em relação ao estado de desempenho. Nesse sentido, esse mecanismo é ortogonal a trabalhos anteriores de última geração, como SAP Hana [Psaroudakis et al. 2016], que propõem dados adaptativos e estratégias de alocação de *thread* entre nós NUMA baseadas em contadores de *hardware* (por exemplo, a intensidade do uso de memória pelas *threads*). No entanto, os resultados mostram que o mecanismo apresentado neste capítulo pode melhorar ainda mais o desempenho de SGBDs relacionais compatíveis com NUMA, como o *SQL Server* [Larson et al. 2012].

O mecanismo apresentado é implementado como um modelo abstrato dos estados de desempenho do SGBD relacional, usando a teoria das Redes de Petri. As metas de desempenho são modeladas como predicados e precisam ser atendidas para acionar a alocação de núcleos de CPU – ou seja, uma *Rede de Petri Predicado/Transição* (PrT) –, de modo a trazer o SGBD relacional para um estado de desempenho estável. A Figura 4.1 apresenta uma visão geral do mecanismo de Rede de Petri no ecossistema de processamento de consultas em máquinas NUMA. A Rede de Petri monitora o uso de recursos das *threads* de trabalho sobre as instalações do *kernel* do SO para decidir pela alocação de núcleos de CPU (por exemplo, *cgroups*, *mpstat*, *numactl*, *likwid*). O local correto para a próxima alocação depende de uma fila de prioridade mantida por

políticas diferentes, chamadas de modos de alocação. Os núcleos alocados são representados em preto.

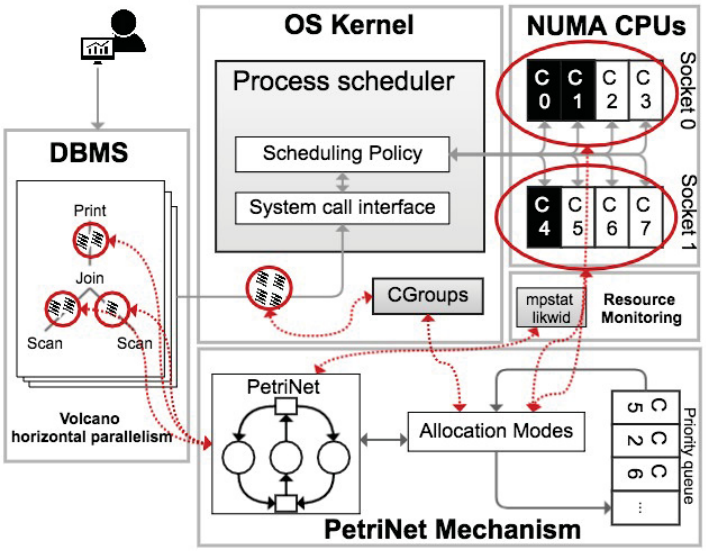


Figura 4.1: Agendador do SO realizando o mapeamento de *threads* de banco de dados suportados pelo mecanismo PetriNet.

No geral, as principais contribuições neste capítulo são as seguintes:

- **Análise da movimentação de dados no NUMA:** distribuir todos os núcleos disponíveis no sistema para SO pode prejudicar o desempenho do processamento de consultas. Sendo assim, utilizando um *microbenchmark*, são discutidos os resultados de uso da largura de banda de interconexão entre os nós NUMA para apresentar a motivação par o estudo do processamento de consultas na arquitetura NUMA.

- **O modelo abstrato:** é proposto um modelo abstrato para alocação de recursos em máquinas NUMA com base nos estados de desempenho do banco de dados. O modelo pode ser facilmente adaptado para alocar memória multi-núcleo ou remota em qualquer SO e SGBD relacional de escolha do usuário.

- **O número ótimo local de núcleos:** apresenta-se o conceito de "número ótimo local de núcleos" para lidar com as cargas de trabalho de processamento de consultas atuais, controlando assim a alocação e liberação de núcleos de processamento ao longo da execução de consultas.

- **Um algoritmo de alocação multi-núcleo adaptativo:** são apresentados diferentes modos de alocação para manter o número ideal local de núcleos e um algoritmo adaptativo que decide o local para a próxima alocação/liberação considerando os endereços de memória acessados. Esses endereços são mantidos em uma estrutura de dados de fila de prioridade.

Este capítulo está organizado da seguinte forma: a próxima seção descreve o problema de movimentação de dados por meio da métrica de interconexão durante o processamento de consultas. A Seção 4.1 discute ainda o modelo abstrato baseado em um *pipeline* de regra-condição-ação. A Seção 4.2 apresenta os principais modos de alocação e os detalhes de implementação. A Seção 4.4 mostra os resultados empíricos e a Seção 4.8 traz as considerações finais.

4.1 IMPACTO DO MOVIMENTO DE DADOS EM ARQUITETURA NUMA

Nesta seção, é utilizado um *microbenchmark* para entender o que acontece ao se mover dados dos nós NUMA para núcleos/*threads* que trabalham nos dados. Esse *microbenchmark* consiste na consulta do *benchmark* TPC-H 06 (Q6), escolhida devido ao seu padrão de acesso a dados com alta localidade espacial [Boncz et al. 2014]. Foram usadas duas versões da Q6: a SQL original e a versão em linguagem *C* codificada manualmente, com processamento vetorizado. Além disso, o mapeamento de *threads*/dados realizado pela política do SO é comparado a uma execução controlada com afinidade de *threads*/dados. Os experimentos foram executados em uma base de dados de 1 GB com variação no número de usuários, em uma máquina *AMD Opteron quad-core* de 4 nós, apresentada no capítulo 2 na Figura 2.11. O objetivo é mostrar que o escalonador do SO não é ideal para o mapeamento de *threads* do SGBD relacional em uma arquitetura NUMA.

SQL:

```
1  SELECT
2     sum(l_extendedprice * l_discount) as revenue
3  FROM
4     LINEITEM
5  WHERE
6     l_shipdate >= date '1997-01-01'
7  AND l_shipdate < date '1997-01-01' + interval '1' year
8  AND l_discount between 0.07 - 0.01 and 0.07 + 0.01
9  AND l_quantity < 24;
```

MAL:

```
1  X_1:= algebra.thetasubselect(l_quantity)
2  X_2:= algebra.subselect(l_shipdate,X_1)
3  X_3:= algebra.subselect(l_discount,X_2,)
4  X_4:= algebra.projection(X_3,l_extendedprice)
5  X_5:= algebra,projection(X_3,l_discount)
6  X_6:= [*](X_4,X_5)
7  X_7:= aggr.sum(X_6)
```

C:

```
1  static double tpch_query6(int lineitemSize, double *__restrict__ p_quanEty, double *__restrict__
2  p_extendedprice, double *__restrict__ p_discount, int *__restrict__ p_shipdate, double
3  sum_revenue, double discount) {
4        for (int i = 0; i < lineitemSize; i++) {
5              if (p_shipdate[i] >= 19970101 && p_shipdate[i] < 19980101) {
6                    if (p_discount[i] >= (discount - 0.01) && p_discount[i] <= (discount + 0.01)) {
7                          if (p_quanEty[i] < 24) {
8                                sum_revenue += p_extendedprice[i] * p_discount[i]; }}}}
9        return sum_revenue; }
```

Figura 4.2: Versão SQL, o plano de consulta escrito em *Monet Assembly Language (MAL)* e código da consulta TPC-H Q6.

A Figura 4.2 mostra as implementações SQL e C da consulta Q6, e o plano de consulta. A consulta Q6 foi implementada utilizando *threads Posix* (*pthreads*) para estabelecer um desempenho próximo ao limite e para analisar se a configuração da afinidade de *pthreads*

conduz o SO a melhorar a localização dos dados. As operações foram executadas em paralelo e a implementação foi constituída pelas colunas que fazem parte da consulta Q6. São utilizadas duas políticas diferentes estipulando a afinidade de *thread* para um ou todos os núcleos utilizando *pthread_setaffinity_np()*. As políticas são constituídas de *dense* – quando as *pthreads* são enviadas para o mesmo nó – e *sparse* – quando as *pthreads* são espalhadas para núcleos em nós NUMA diferentes. Esses modos de afinidade seguem a ideia de alocação incremental apresentada em [Porobic et al. 2012; Dominico et al. 2017]

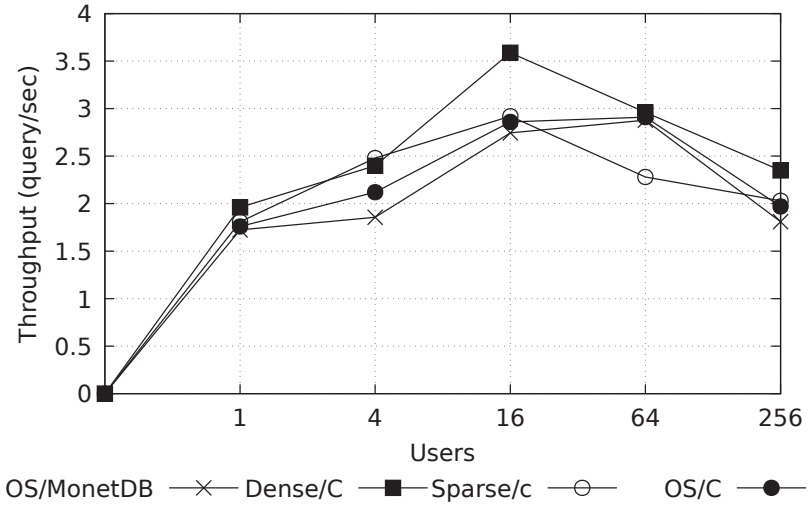


Figura 4.3: Consultas por segundo na execução do TPC-H Q6 com um número crescente de clientes concorrentes.

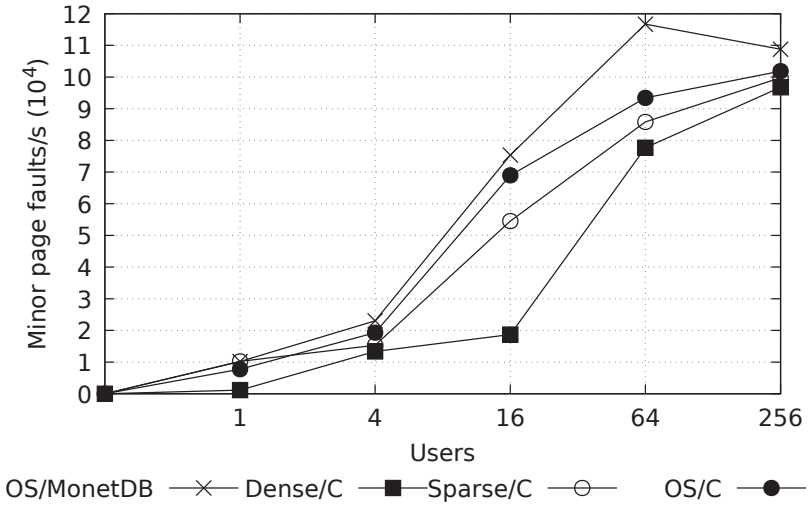


Figura 4.4: *Minor page faults* na execução do TPC-H Q6 com um número crescente de clientes concorrentes.

As Figuras 4.3, 4.4 e 4.5 mostram os experimentos com o MonetDB (OS/MonetDB) e a versão codificada em linguagem *C* (OS/C), além do mapeamento para o código *C* usando as afinidades *sparse* e *dense* (*Sparse*/C e *Dense*/C respectivamente). É possível observar que o tráfego de interconexão aumenta conforme o aumento no número de usuários simultâneos – com um número maior de usuários, o SO obriga um equilíbrio de carga. No entanto, o SO apresenta mais facilidade em encontrar a afinidade de dados na execução do código *C* em comparação com a execução da consulta no MonetDB. A versão SQL gera diversas *threads* para cada operador do

plano de consulta, criando um sistema de iteração mais complexo se comparado ao código em *C*, que gera as *threads* a partir de um único programa. Consequentemente, essa diferença reflete-se no desempenho da consulta na versão SQL, que diminui, e no número crescente de falhas de páginas (*minor page faults*) (Figura 4.3 e 4.4). O uso de interconexão entre as versões SQL e C varia de 100×, com um único cliente, a 8×, com 256 clientes (Figura 4.5).

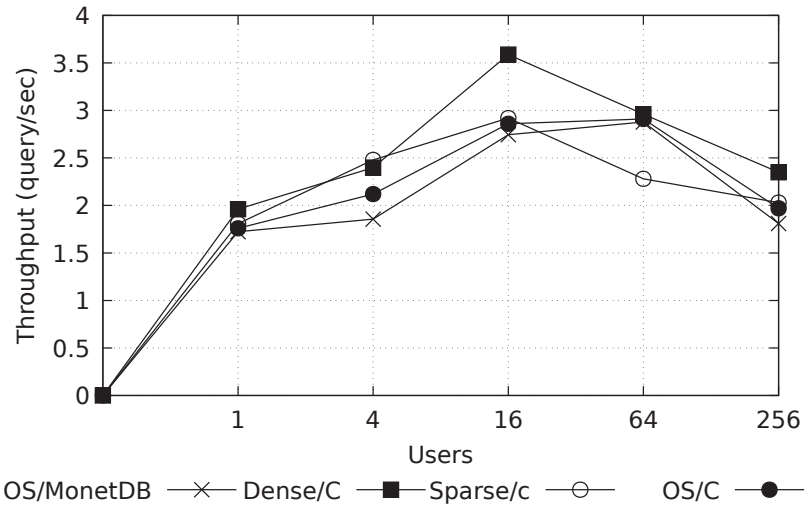


Figura 4.5: *HT traffic* na execução do TPC-H Q6 com um número crescente de clientes concorrentes.

A quantidade de falhas de páginas foi analisada para se estudar o volume de dados movidos entre os nós NUMA. Uma falha de página ocorre quando o SO aloca memória para uma *thread* no nó local no início da execução e os mesmos dados buscados pela *thread* já foram utilizados por outra *thread* em um nó remoto. Ao realizar a busca de um dado já utilizado, ocorre uma nova falha de página e um acesso remoto com custo adicional de movimentação dos dados.

O número de falhas de páginas (*minor page faults*) aumenta conforme o número de usuários concorrentes aumenta. Além disso, observa-se que a vazão (q/s) diminui após 16 usuários concorrentes. Na execução OS/MonetDB o uso de banda de interconexão (*HT traffic*) apresentou um crescimento com todos os núcleos em uso, indo de 840 MB/s de tráfego de interconexão com 16 usuários concorrentes para 8 GB/s com 256 usuários concorrentes (Figura 4.5). Com mais usuários ou mais núcleos, o SO tenta manter o equilíbrio de carga mesmo sem possuir um mecanismo para auxiliar na detecção de um bom mapeamento das *threads*.

O mapeamento das *threads* pelo SO com a execução da consulta Q6 foi analisado com um único usuário no MonetDB. A Figura 4.6 mostra a migração das *threads* geradas pelo MonetDB. As cores estão representando os diferentes nós NUMA e os tons indicam os núcleos da CPU no mesmo nó. Nota-se o esforço do SO para manter o equilíbrio de carga: as *threads* são migradas constantemente entre os nós NUMA durante o tempo de execução. A política de mapeamento de *threads* padrão do SO (*first touch*) não é consciente da arquitetura NUMA, o que, consequentemente, traz um custo adicional de latência aos núcleos que realizam acessos à memória remota.

Para entender o paralelismo do MonetDB, foi utilizado o *Tomograph* [Gawade e Kersten 2013] com o objetivo de verificar a execução de cada *thread* ao executar a consulta Q6. A Figura 4.7 mostra as *threads* pelo *Tomograph*. O primeiro operador do plano de consulta representado pela barra cinza é o *thetasubselect*, executado por 15 *threads* que acessam paralelamente partições da coluna *l_quantity* implementada internamente como um vetor chamado BAT (*Binary Association Table*). O acesso paralelo pressiona o escalonador do SO a encontrar uma boa localidade NUMA,

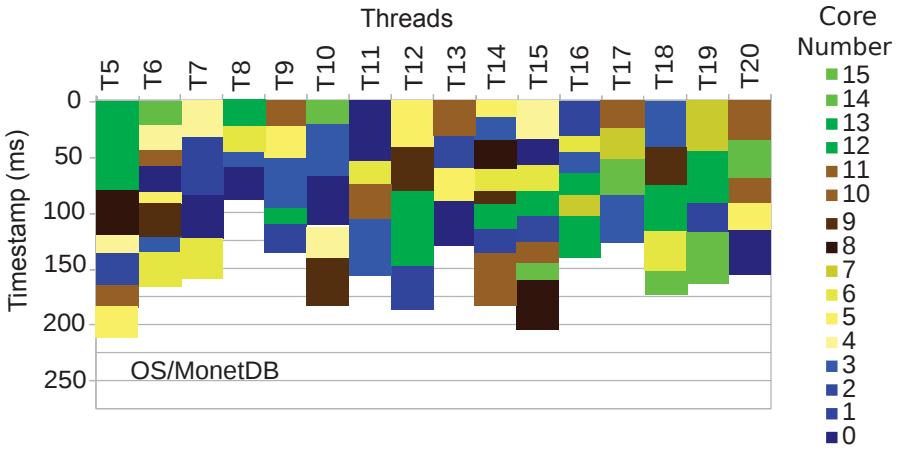


Figura 4.6: Avaliação da vida útil e da migração das *threads* geradas pela Q6 entre múltiplos núcleos em uma execução com cliente único com todos os 16 núcleos disponíveis.

movimentando as *threads* em todos os nós, o que se reflete nos resultados observados de falhas de páginas e tráfego de interconexão. Já o acesso paralelo a muitas partições de dados pressiona o escalonador do SO para encontrar uma boa localidade NUMA, refletindo-se nos números observados de perdas de *cache* e tráfego de HT.

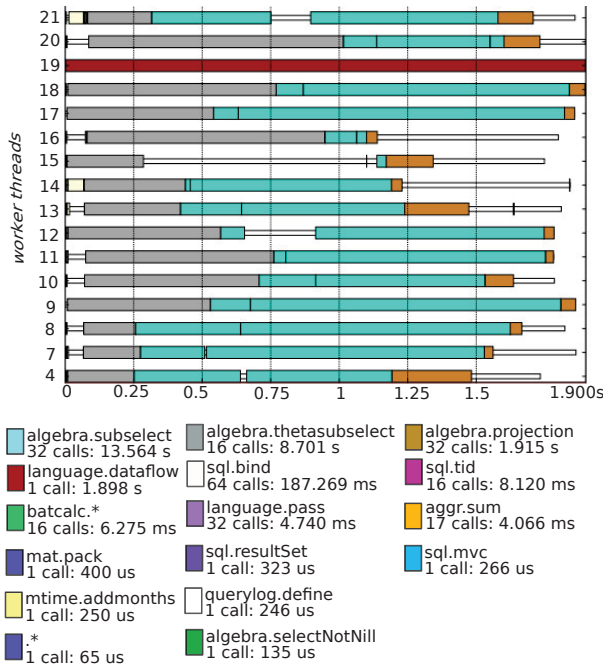


Figura 4.7: Captura de tela do recurso *Tomograph* mostrando as 16 *threads* geradas pelo MonetDB para executar a consulta Q6.

Na próxima seção, é apresentado o mecanismo de alocação dinâmica de núcleos de processamento para cargas de trabalho de processamento de consultas. Tal mecanismo tem dois objetivos: minimizar a movimentação de dados e reduzir os acessos remotos entre os nós NUMA durante o processamento de consultas pelo SGBD relacional. O processamento de consultas fornece padrões diferentes de consumo de CPU e memória em uma mesma carga de trabalho [Gawade et al. 2016]. Portanto, existem pressões constantes no SO para atualizar a alocação das *threads* nos nós NUMA para manter o equilíbrio de carga e a afinidade dos dados.

O mecanismo de alocação dinâmica precisa reagir às flutuações de CPU e consumo de memória principal, auxiliando o SO a alocar às *threads* com o menor impacto negativo sobre o desempenho. Nesse contexto, o mecanismo deve facilmente escalar e encontrar dinamicamente o número de núcleos para lidar com a carga de trabalho corrente.

4.2 MODELO ABSTRATO DE ALOCAÇÃO DINÂMICA DE NÚCLEOS

O mecanismo de alocação de núcleo implementa um modelo abstrato baseado em transições de estado de desempenho do banco de dados em execução. O modelo abstrato utilizado é uma Rede de Petri predicado/transição PrT, que fornece a capacidade para definir condições de desempenho (predicados) utilizando informações de uso de recursos computacionais pelas *threads* do banco de dados. As Redes de Petri são amplamente utilizadas na literatura para modelar propriedades de desempenho de sistemas dinâmicos e concorrentes [Desel e Reisig 2015; He 1996].

A PrT é um grafo orientado bipartido que representa o fluxo de uma marcação ao redor da topologia da rede. Ela possui dois tipos de nós, os lugares e as transições, que são conectados com arcos. Uma marca representa a existência ou não de um estado e a quantidade de uma determinada entidade. No modelo usado nesta tese, a marca representa a quantidade de núcleos disponíveis para alocação e o uso de recurso computacional.

A PrT é formalmente definida como uma tupla $PrT = \{P, T, F, R, M\}$. $P \cap T = \emptyset$ é um conjunto finito disjunto que define lugares e transições. A estrutura essencial da rede é o subdomínio $\{P, T, F\}$. Os lugares ($p \in P$) são os estados do banco de dados durante a execução. As transições ($t \in T$) definem as condições de alternar os estados de desempenho baseado nas variáveis $v = \{u, n_{alloc}, n_{total}\}$, em que u representa a média de uso de recursos (ex. carga de CPU em porcentagem), $n_{alloc} \in \mathbb{N}$ representa o número de núcleos alocados e n_{total} representa o total de núcleos disponíveis no *hardware*.

A Figura 4.8 ilustra as definições utilizadas para o modelo do mecanismo de alocação de núcleos. Para exemplificar, é considerada a execução da consulta Q6 do *benchmark* TPC-H com um único usuário. Quando a carga de trabalho das *threads* sobe, a transição é imediatamente disparada entre estados de desempenho para alocar núcleos para o SO. Similarmente, quando a carga de trabalho cai, os núcleos são liberados.

O conjunto F define os arcos $< p_i, t_j >$ ou $< t_j, p_i >$ entre os lugares e transições $F \subseteq (P \times T) \cup (T \times P)$. Duas funções definem a relação de fluxo $Pre(P \times T)$ e $Post(T \times P)$. A função de Pre define o lugar de saída para uma transição de entrada com um arco $< p_i, t_j >$ se e somente se $Pre(p_i, t_j) \neq 0$. A função $Post$ define a transição de saída para um lugar de entrada com um arco $< t_j, p_i >$ se e somente se $Pre(t_j, t_i) \neq 0$.

O subdomínio R, M define a semântica do modelo, onde $R : T \rightarrow < oper, bool > (X)$ é um mapeamento de restrições bem definido, que associa cada transição T a uma fórmula lógica de primeira ordem. A lógica define as condições na rede para o disparo de uma transição. A marcação inicial é definida como M: $P \rightarrow \mathbb{N}$ com a distribuição inicial das marcas nos lugares. A função $M(p)$: $u \rightarrow \mathbb{N}$ define quantas marcas um determinado lugar possui.

Os lugares no modelo de alocação de núcleos são definidos como $P = \{Stable, Idle, Overload, Provision, Checks\}$ e as transições como $T = \{t0, ..., t7\}$. Os três primeiros lugares representam os estados de desempenho do banco de dados e os dois últimos os estados complementares, onde $Checks$ é atualizado de forma assíncrona com o uso corrente de recursos e $Provision$ com a quantidade de núcleos em uso. Para facilitar o entendimento do modelo, a rede foi dividida em três sub-redes, considerando os três estados de desempenho do banco de dados ($Stable, Idle, Overload$).

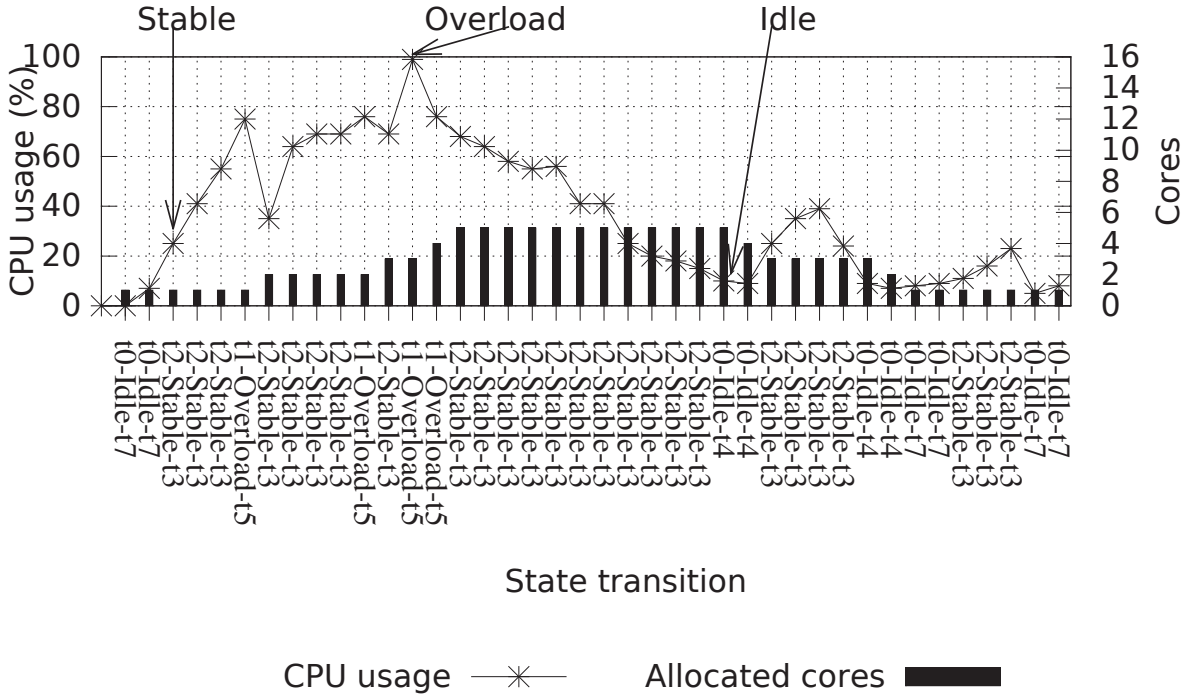


Figura 4.8: Transições de estado da consulta Q6 e a alocação de núcleos. O eixo X representa as transições disparadas temporalmente. O eixo Y, na extremidade esquerda, é o uso da CPU (%). O eixo Y, na extremidade direita, é o número de núcleos alocados.

Os limiares de uso de recurso foram definidos para estabelecer quando é necessária a alocação/remoção de um núcleo. Para exemplificar o modelo, é considerado o uso de CPU (em porcentagem). Os limiares são definidos como th_{min} o limiar inferior e th_{max} o limiar superior. Se os limiares de desempenho forem respeitados, o banco de dados se encontrará no estado estável ($Stable$) e somente o monitoramento é necessário. Uma PrT pode ser representada utilizando uma notação matricial nomeada de matriz de incidência que representa o relacionamento entre as transições e lugares.

Uma matriz de incidência prévia (Pre) é representada pelas dimensões mxn em que n é o numero de lugares – no caso do modelo desenvolvido os estados – e m o número de transições. Os elementos da matriz são os pesos dos arcos que interligam um determinado lugar com uma transição. A ligação de um lugar e uma transição somente existe se o valor é diferente de 0, o que torna um lugar de incidência da transição. Já a matriz de incidência posterior $Post$ tem dimensão as mesas dimensões da matriz Pre, cada elemento da matriz corresponde ao peso do arco que liga uma transição com um lugar, os elementos diferentes de 0 indicam a ligação da transição para o lugar, sendo a coluna a transição e o lugar a linha. As matrizes Pre e $Post$ representam as pré e pós-condições de todas as transições da rede.

Através da subtração das matrizes Pre e $Post$ define-se a matriz de incidência A^T. A matriz de incidência possibilita a análise algébrica do comportamento dinâmico das redes de Petri através do disparo das transições. Na matriz A^T analisamos a acessibilidade de um determinado lugar através da marcação inicial de uma Rede de Petri. A Figura 4.9 apresenta a estrutura do modelo como uma matriz de incidência $A^T = Post - Pre$.

A marcação inicial do conjunto M em cada estado é definida como: $m_0(Checks) = \{0, \ldots, 100\}$, $m_0(Idle) = m_0(Stable) = m_0(Overload) = \{0\}$ e $m_0(provision) = \{r\}$ (i.e., $n_{alloc} == 1$ número padrão de núcleos alocados inicialmente). Para exemplificar a explicação, a carga de trabalho corrente utiliza 3 de 16 núcleos ($n_{alloc} == 3$ e $n_{total} == 16$) de uma máquina

$$
\begin{array}{c} Post \\ p_0 \\ p_1 \\ p_2 \end{array}
\begin{array}{c} \begin{array}{ccc} t_0 & t_1 & t_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array}
-
\begin{array}{c} Pre \\ p_0 \\ p_1 \\ p_2 \end{array}
\begin{array}{c} \begin{array}{ccc} t_0 & t_1 & t_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}
= A^T
\begin{array}{c} \begin{array}{ccc} p_0 & p_1 & p_2 \end{array} \\ \begin{pmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \end{array}
$$

Figura 4.9: A transposição A^T orienta a relação de fluxo com base nas pré-condições Pre e pós-condições $Post$.

AMD Opteron e os limiares de uso de CPU baseados em regras da literatura [Minhas et al. 2012] ajustadas com experimentos empíricos. Os limiares são definidos como $th_{min} = 10$ e $th_{max} = 70$.

4.2.1 Sub-rede Overload

A sub-rede $Overload$ modela a alta carga de uso de CPU quando são necessários mais núcleos de processamento. O estado de $Overload$ é esperado para cargas de trabalho de processamento de consultas. Geralmente, o SO trabalha com todos os núcleos de processamento disponíveis, enquanto no modelo somente são disponibilizados núcleos de CPU sob-demanda quando detectado o uso de CPU alto. O objetivo final é fornecer núcleos conforme o comportamento da carga de trabalho, considerando a carga e também o compartilhamento de dados entre as *threads*.

$$
\begin{array}{c} Post \\ Checks \\ Overload \\ Provision \end{array}
\begin{array}{c} \begin{array}{cc} t_1 & t_5 \end{array} \\ \begin{pmatrix} 0 & u \\ n_a & 0 \\ 0 & n_a \end{pmatrix} \end{array}
-
\begin{array}{c} Pre \\ Checks \\ Overload \\ Provision \end{array}
\begin{array}{c} \begin{array}{cc} t_1 & t_5 \end{array} \\ \begin{pmatrix} u & 0 \\ 0 & n_a \\ n_a & 0 \end{pmatrix} \end{array}
= A^T
\begin{array}{c} \begin{array}{ccc} Checks & Overload & Provision \end{array} \\ \begin{pmatrix} -u & n_a & -n_a \\ u & -n_a & n_a \end{pmatrix} \end{array}
$$

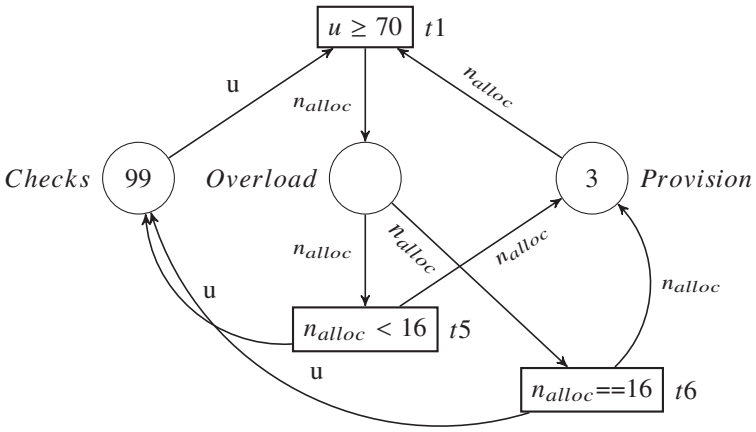


Figura 4.10: Transição de marcas na sub-rede $Overload$ para alocar um núcleo com $u = 99\%$, $n_{alloc} = 3$ de $n_{total} = 16$ núcleos e $th_{max} = 70\%$.

Para exemplificar a sub-rede $Overload$, a Figura 4.10 representa o disparo de t_1 quando a carga de CPU é $u = 99\%$ com $th_{max} = 70\%$, considerando 3 núcleos adicionados de 16 núcleos. Em seguida, t_5 aciona a transição para $Provision$ para alocar mais um núcleo se ainda houver núcleos disponíveis para serem alocados (i.e., $n_{alloc} < n_{total}$). Nesse caso, t_6 somente será disparada se $n_{total} == 16$, ou seja, não há mais núcleos disponíveis, forçando somente o monitoramento enquanto o estado não é modificado. A Figura 4.8 mostra esta transição como

"$t_1 - Overload - t_5$", durante a execução da consulta Q6. Em paralelo, $t5$ dispara para $Checks$ para validar a carga de CPU, que desceu para $u = 68\%$.

4.2.2 Sub-rede Idle

A sub-rede $Idle$ modela o banco de dados em estado ocioso, quando é necessário remover núcleos de processamento. A condição para disparar $t0$ é o baixo uso de CPU em vários núcleos. Em seguida, $t4$ dispara se ainda existirem núcleos para serem removidos (i.e., $n_{alloc} > 1$). Caso contrário, dispara a transição $t7$ para $Checks$, aguardando atualizações adicionais nas variáveis. A transição $t7$ limita o menor número de núcleos no sistema.

Para exemplificar, a Figura 4.11 mostra o disparo de $t0$ quando o uso de CPU está em $u = 10\%$ e o limiar inferior é de $th_{min} = 10\%$, com 5 núcleos provisionados. Em seguida, t_4 dispara a transição para $Provision$ para remover um núcleo ($n_{alloc} = 4$). A Figura 4.8 apresenta essa transição como "$t_0 - Idle - t_4$". Concorrentemente, t_4 dispara para $Checks$ para validar novamente a carga de CPU, que desce para 5% após a transição.

Segundo o exemplo, a matriz de incidência da sub-rede $Idle$ mostra as pré-condições de disparo das transições como $Checks - t_0, Provision - t_0, Idle - t_4$. O arco $Idle - t_7$ não está na matriz Pré, pois a validação não permite pós-condição, ou seja, neste exemplo a transição $t7$ não dispara. Na pós-condição, estão $\{t_4 - Checks, t_4 - Provision, t_0 - Idle)\}$. A matriz de incidência é representada como:

$$
\begin{array}{lc} Post & \begin{array}{cc} t_0 & t_4 \end{array} \\ \begin{array}{l} Checks \\ Idle \\ Provision. \end{array} & \begin{pmatrix} 0 & u \\ n_a & 0 \\ 0 & n_a \end{pmatrix} \end{array} - \begin{array}{lc} Pre & \begin{array}{cc} t_0 & t_4 \end{array} \\ \begin{array}{l} Checks \\ Idle \\ Provision. \end{array} & \begin{pmatrix} u & 0 \\ 0 & n_a \\ n_a & 0 \end{pmatrix} \end{array} = A^T \begin{array}{c} \begin{array}{ccc} Checks & Idle & Prov. \end{array} \\ \begin{pmatrix} -u & n_a & -n_a \\ u & -n_a & n_a \end{pmatrix} \end{array}
$$

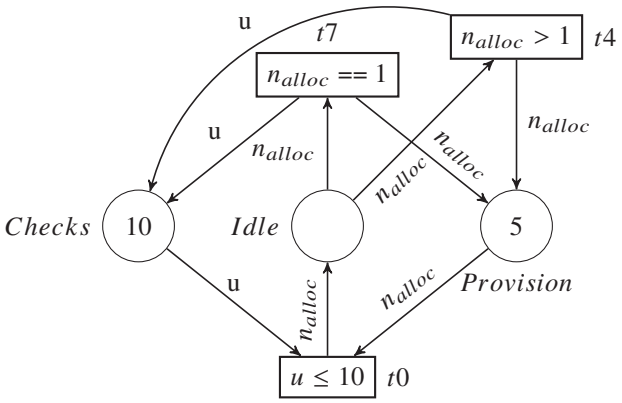


Figura 4.11: Marcação através da transição na sub-rede $Idle$ para liberar um dos 5 núcleos de CPU com $u = 8\%$ e $th_{min} = 10\%$.

4.2.3 Sub-rede Stable

A sub-rede $Stable$ modela o banco de dados em estado estável, em que o número corrente de núcleos é necessário para atender à carga de trabalho em execução. O uso de CPU se encontra entre os limiares máximo e mínimo. A matriz pré para a sub-rede $Overload$ apresenta as conexões entre "$Checks - t_2$" e "$Stable - t_3$". Por exemplo, a Figura 4.8 representa o disparo de t_2 para uma carga de CPU de $u = 40\%$ com $th_{min} = 10$ e $th_{max} = 70$.

A matriz *Post* para a sub-rede *Stable* mostra os arcos de conexão entre "$t_2 - Stable$" e "$t_3 - Checks$". A matriz de incidência para a sub-rede *stable* mostra o efeito do disparo de t_2 e t_3. A primeira linha apresenta que o disparo remove uma marca de *Checks* e adiciona em *Stable*. A segunda linha mostra que o disparo de t_3 remove a marca de *Stable* e adiciona em *Checks*. Não é necessária nenhuma alocação de núcleo neste cenário. A matriz de incidência é apresentada como segue:

$$\begin{array}{lc} Post & t_2\ t_3 \\ Checks \\ Stable \end{array} \begin{pmatrix} 0 & u \\ u & 0 \end{pmatrix} - \begin{array}{lc} Pre & t_2\ t_3 \\ Checks \\ Stable \end{array} \begin{pmatrix} u & 0 \\ 0 & u \end{pmatrix} = A^T \begin{pmatrix} Checks & Stable \\ -u & u \\ u & -u \end{pmatrix}$$

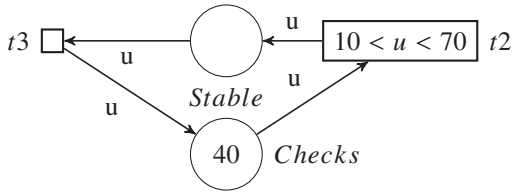


Figura 4.12: Marcação através da transição na sub-rede *Stable* com $u = 40\%$ e limiar entre 10% e 70%.

4.3 ALOCAÇÃO DE NÚCLEOS DE PROCESSAMENTO

O objetivo da alocação dos núcleos é encontrar o número ideal de núcleos de processamento que atenda a uma determinada carga de trabalho, evitando a subutilização ou utilização total dos núcleos. Além disso, o mecanismo deve considerar a alocação de dados nas regiões NUMA, alocando núcleos próximo a nós NUMA específicos (próximo ao espaço de endereço em que ocorreu o "*first touch*"), evitando a movimentação de dados e migração de *threads*.

4.3.1 Número Ótimo de Núcleos Local

O Número Ótimo de Núcleos Local (*Local Optimum Number of Cores* — LONC) é encontrado quando a carga de uso de CPU está no estado estável (*Stable*). É um número local porque é utilizada a média aritmética de carga de CPU das *threads* ativas do banco de dados.

Formalmente, o LONC é definido como:

$$\forall\, w\ \exists\, n_{alloc} | (th_{min} < u < th_{max}) \wedge p(n_{alloc}) \geq p(n_{total}) \tag{4.1}$$

onde:

- w é a carga atual das *threads* do banco de dados;
- u é a média do uso de recursos pelas *threads* do banco de dados;
- n_{alloc} é a quantidade de núcleos de processamento alocados ($n_{alloc} \leq n_{total}$);
- n_{total} é o número de núcleos de processamento disponíveis;
- $p(x)$ é a função de desempenho –representa o desempenho da carga de trabalho com um número específico de núcleos –, em que x assume o número de núcleos de processamento n_{alloc} ou n_{total}.

Para toda carga de trabalho w, existe um certo número de núcleos de processamento n_{alloc}. A carga de uso de CPU está entre os limiares mínimo e máximo, em que o desempenho do banco de dados $p(n_{alloc})$ é igual, ou melhor do que a desempenho $p(n_{total})$ com todos os núcleos disponíveis no *hardware*. A função de desempenho $p(x)$ depende dos contadores do sistema fornecidos pelo SO e pelo SGBD relacional.

Na implementação, a função de desempenho utiliza a funcionalidade do SO para manter os contadores atualizados no modelo abstrato e melhorar a precisão da alocação de recursos. Inicialmente, utiliza-se o $cgroups$, que é um recurso do *kernel* do SO *Linux*, para isolar as *threads* do SGBD relacional em grupos hierárquicos específicos (modos de alocação). Utilizando o $cgroups$, são agrupados os números de *thread identifier* (TIDs) das *threads* do banco de dados para monitorar sua execução e limitar seus recursos disponíveis (por exemplo, núcleos). Uma fila de prioridades é implementada com informações de uso de recursos NUMA, com o núcleo de execução e espaço de endereço. O *MPstat* é utilizado para coletar informações de carga de CPU dos TIDs. Por fim, por meio da ferramenta Likwid [Treibig et al. 2010], é possível coletar obter falhas de $cache$ em L3 (L3 *cache data misses*), a largura de banda nos $links$ de interconexão NUMA e o tráfego de memória.

A carga de CPU para o cálculo do LONC também considera outros processos do SGBD relacional, além das *threads* da execução da consulta. Quando as operações utilizam completamente a CPU, um novo núcleo de processamento é alocado para manter o uso estável e impedir que as *threads* do SGBD relacional aguardem para começar a executar. O resultado esperado é alcançar um melhor tempo de resposta, com mais acessos à *cache* e menos acessos à memória remota para o processamento de consultas. A Figura 4.8 mostra a execução da $Q6$ com o suporte do mecanismo proposto neste trabalho. Exemplifica-se o comportamento do mecanismo quando a carga sobe para 99% (transição $t_1 \rightarrow t_5$), os núcleos de processamento são liberados quando a carga cai para 8% (transição $t_0 \rightarrow t_4$).

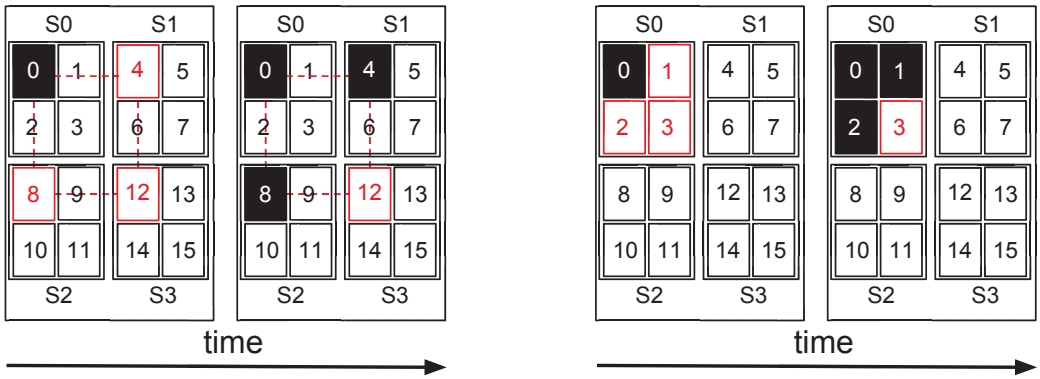
4.3.2 Modos de Alocação Multi-núcleo

A alocação de núcleo é dividida em três modos principais de alocação para atender às necessidades de diferentes modelos de *threads* do SGBD relacional. Uma *thread* pode apresentar a necessidade de mais memória compartilhada entre as *threads* e ser alocada em um mesmo nó. Em contraste, o modelo pode apresentar pouca memória compartilhada entre as $threads$ e alocá-las em nós diferentes para evitar a concorrência de memória entre elas. Utilizando a descrição da máquina NUMA da Figura 1.1, exemplificam-se os modos de alocação tratados a seguir.

4.3.2.1 Modo de alocação Denso e Esparso

Os modos de alocação denso ($Dense$) e esparso ($Sparse$) executam regras para a alocação dinâmica de núcleos. A função de alocação é simples e mapeia um nó NUMA com índice i para seu núcleo j, sendo definida pelo núcleo $core(i, j) = d.i + j$, onde $1 \leq j \leq d$. Nessa função, d é uma constante para representar uma máquina de nó d-ário, mostrada como $d = 4$ pela Figura 4.13 para representar a máquina AMD Opteron de quatro nós.

O modo Esparso itera sobre i, j para alocar um núcleo por vez em um nó NUMA diferente. O modo Denso itera sobre i, j para alocar um núcleo por vez em um mesmo nó NUMA. A Figura 4.13 (a) mostra o modo de alocação esparso e a Figura 4.13 (b) mostra o modo denso, ambos ao longo do tempo.



(a) Modo Esparso (b) Modo Denso

Figura 4.13: Modo Denso e Esparso ao longo do tempo – apenas as caixas pretas (i.e., núcleos).

4.3.2.2 *Modo Adaptativo de Prioridade*

O objetivo do modo de alocação Adaptativo é alocar núcleos baseado nas informações de uso de memória. Uma fila de prioridade é usada para indicar o nó NUMA com a maior/menor quantidade de memória alocada (na prioridade superior/inferior) e o modelo aloca/libera um núcleo próximo a esse espaço de endereço. Cada entrada da fila de prioridade mantém os TIDs das threads ativas com seus espaços de endereço e o número de páginas por nó NUMA.

Cada entrada da fila de prioridades mantêm os TIDs das *threads* do banco de dados ativas, com seus espaços de endereço e o número de páginas por nó NUMA. No fluxo de trabalho do modo adaptativo, o número de páginas por nó NUMA é registrado em um contador para reconhecer o nó com a maior quantidade de páginas. O nó NUMA com maior prioridade é aquele com o maior valor de contador. Analogamente, o nó NUMA com o menor valor de contador é aquele com menor prioridade. Se for necessário um novo núcleo de processamento, ele será alocado no nó NUMA de maior prioridade. Se um núcleo de processamento precisar ser liberado, um núcleo no nó NUMA com a menor prioridade será removido.

4.3.3 Limiares dinâmicos

Inicialmente, o mecanismo abstrato proposto é baseado em limiares com valores estáticos, utilizando informações da literatura e ajustes realizados empiricamente. Entretanto, para fornecer um mecanismo completamente dinâmico, apresenta-se uma técnica que utiliza informações de uso de recursos para designar limiares para cada etapa da carga de trabalho. Com esse objetivo, é utilizado um método baseado em análises estatísticas, nomeado de Controle Estatístico de Processo (CEP) [Montgomery 2007].

O CEP é um conjunto de técnicas estatísticas que são utilizadas para monitoramento de um processo na produção. O objetivo é auxiliar na prevenção e detecção de problemas que influenciem a qualidade final do produto. Por meio de um gráfico de controle, é possível verificar alterações nos parâmetros do processo. Utilizando dados estatísticos, o gráfico de controle estima os limites de controle superior (LSC) e inferior (LIC), semelhante aos limiares de uso de recursos propostos. O gráfico de controle é baseado em média amostral e desvio padrão [Montgomery 2007].

No gráfico de controle, se os pontos amostrais estiverem entre as linhas de controle, o processo é dito sob controle. Caso contrário, está fora de controle e uma ação deve ser tomada. Quando o uso de recursos está fora do estado estável (entre os limiares), similarmente, a ação do nosso mecanismo pode ser adicionar ou remover núcleos. Os limites de controle são calculados

conforme apresentado nas Equações 4.2 e 4.3 – respectivamente, limite inferior e limite superior. Os dados utilizados pelo cálculo são o desvio padrão da amostra, a média (u) e o tamanho da amostra (n).

$$LIC = u - 3(\alpha/\sqrt{n}) \tag{4.2}$$

$$LSC = u + 3(\alpha/\sqrt{n}) \tag{4.3}$$

Os cálculos são realizados em amostras com distribuição normal, para definir os limiares e detectar as anomalias durante o processo. No contexto proposto neste trabalho, para utilizar o CEP, é necessário considerar o sistema em modo estável e ignorar amostras quando o sistema estiver em estado de sobrecarga ou ocioso. Nesse sentido, o valor de uso de recurso utilizado precisa ser coletado com o sistema estável e deve ser considerada a quantidade de núcleos já alocados para o cálculo realizado por núcleo. A amostra tem tamanho variado e a hipótese é que os limiares inferiores e superiores com diferentes quantidades de núcleos terão um padrão com pouca alteração.

O cálculo dos limiares dinâmicos é realizado em paralelo com a execução da PrT e são informadas as variações quando um novo núcleo é alocado. Inicialmente, foi definido 70% para o limiar máximo e 10% para o limiar mínimo. Ao longo do monitoramento do mecanismo proposto nesta tese, esses valores tornam-se dinâmicos, baseados na carga de trabalho em determinado tempo de execução.

4.4 ANÁLISE EXPERIMENTAL

Foi implementado um protótipo em linguagem C e foram executados os experimentos no SO Debian Linux 8 “Jessie”, com *kernel* 3.16.0-4-amd64. O mecanismo de alocação de núcleos é comparado ao escalonamento do SO das *threads* geradas pelo SGBD relacional MonetDB (v11.25.5). O mecanismo é avaliado com o SGBD relacional *SQL Server*, que possui reconhecimento da arquitetura NUMA (*v2017 Developer RC2*). O *SQL server* implementa um modelo de *threads* similar, baseado no *pipeline*. Já o MonetDB implementa acesso paralelo semelhante às estruturas de dados de coluna (ou seja, BAT ou vetores) [Larson et al. 2012]. Em ambos os experimentos, não existiu nenhuma configuração adicional nos SGBDs relacionais ou alteração no *SO*.

Os experimentos foram realizados em uma máquina formada por 4 nós *AMD Opteron* 8387, executando a 2,8 GHz. Cada nó *Opteron* é formado por quatro núcleos com $cache\ L1$ privado (64 KB) e um $cache$ L3 compartilhado (6 MB). Os nós NUMA são interconectados pelo *link* HT $3x$ largura de banda agregada máxima de 41.6 GB/s (32 bits). Esta máquina inclui 64 GB de memória principal DDR-2 e disco SATA de 1.8 TB. Sem o suporte do mecanismo de alocação de núcleos, todos os 16 núcleos ficaram disponíveis para o SGBD relacional. Os resultados são provenientes de mais de 10 execuções para cada modo de alocação.

Os valores de limiares iniciais estáticos foram definidos em $th_{min} = 10\%$ e $th_{max} = 70\%$, baseados na literatura [Minhas et al. 2012]. Ajustados com experimentos empíricos, esses valores são mantidos em parte dos experimentos. A sobrecarga do mecanismo de alocação de núcleo foi medida considerando o fluxo de marcas em uma matriz 5×8 para acionar uma transição em no modelo abstrato. No modelo denso, o fluxo de marcas leva, em média, $0,017$ segundos, o modo esparso leva $0,021$ segundos e o modo adaptativo $0,031$ segundos. A média da carga de CPU, considerando a mudança de estados, é inferior a 1%.

SQL:

```
1  SELECT
2     sum(l_extendedprice * l_discount) as revenue
3  FROM
4     LINEITEM
5  WHERE
6     l_shipdate >= date '1997-01-01'
7  AND l_shipdate < date '1997-01-01' + interval '1' year
8  AND l_discount between 0.07 - 0.01 and 0.07 + 0.01
9  AND l_quantity < 24;
```

MAL:

```
1  X_1:= algebra.thetasubselect(l_quantity)
2  X_2:= algebra.subselect(l_shipdate,X_1)
3  X_3:= algebra.subselect(l_discount,X_2,)
4  X_4:= algebra.projection(X_3,l_extendedprice)
5  X_5:= algebra,projection(X_3,l_discount)
6  X_6:= [*](X_4,X_5)
7  X_7:= aggr.sum(X_6)
```

Figura 4.14: Versão SQL da consulta Q6 e o plano de consulta escrito em *Monet Assembly Language (MAL)*.

4.5 ANÁLISE DA CONSULTA Q6 DO TPC-H

Nesta seção, é investigado o impacto do mecanismo ($Adaptive$), com o escalonamento normal do SO ($OS/MonetDB$) e com os dois modos de alocação suportados pelo mecanismo de alocação de núcleo. O foco dessa análise é o operador *thetasubselect* da consulta Q6 que pode ser observada na Figura 4.14, pelo fato de movimentar uma grande quantidade de dados antes de outros operadores no plano, sendo a coluna $l\_quantity$ com distribuição uniforme e 45% de seletividade. Esse cenário beneficia o modo denso e o adaptativo porque se espera que o planejador do SO mapeie as *threads* próximas ao espaço de endereço de memória da coluna.

4.5.1 Impacto do Agendamento

Nesta seção, são apresentados os impactos dos modos de alocação no mapeamento das *threads* à medida que a concorrência aumenta. O protocolo de execução segue a descrição de [Psaroudakis et al. 2015], porém limitado a 256 usuários concorrentes executando a versão com o operador *thetasubselect* da consulta $Q6$ em 1 GB.

A Figura 4.15 mostra as métricas de desempenho relacionadas à arquitetura NUMA. O *throughput* da consulta atingiu seu pico em 64 usuários simultâneos, mas as melhorias de desempenho ocorreram em todas as escalas (Figura 4.15 (a)). O modo adaptativo ($Adaptive$) apresentou maior eficiência e desempenho, alcançando 25% mais *throughput* de consulta do que o escalonador do SO. O uso de CPU (*CPU Load*) e as tarefas ($Tasks$) permaneceram semelhantes em todos os modos (Figura 4.15 (a) e (c) respectivamente). No entanto, o esforço do SO para manter o equilíbrio de carga resultou em 46% mais tarefas roubadas (*Stolen Tasks*) no experimento do OS/MonetDB do que o modo adaptativo proposto neste trabalho (Figura 4.15 (d)).

A Figura 4.16 mostra o uso de memória para indicar do impacto das tarefas roubadas (*Stolen Tasks*). Observa-se que o mecanismo de alocação de núcleos diminuiu os erros de *cache* em L3 (*L3 cache misses*) em 43%, melhorando o *throughput* de memória em 27% (Figuras 4.16 (a)

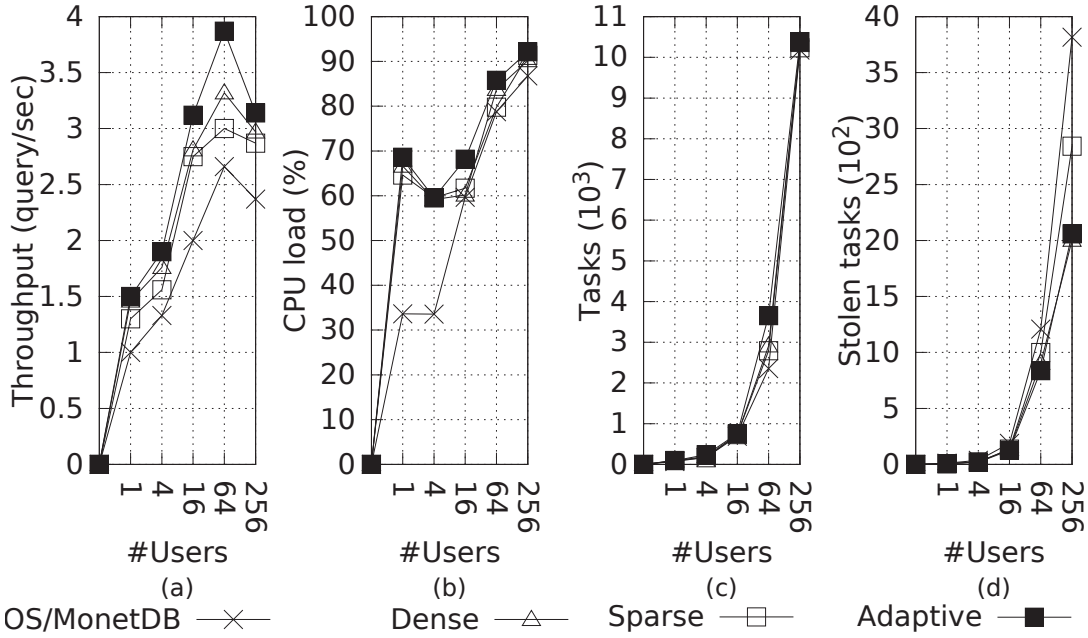


Figura 4.15: Métricas de desempenho ao processar um número crescente de usuários simultâneos executando o *thetasubselect*.

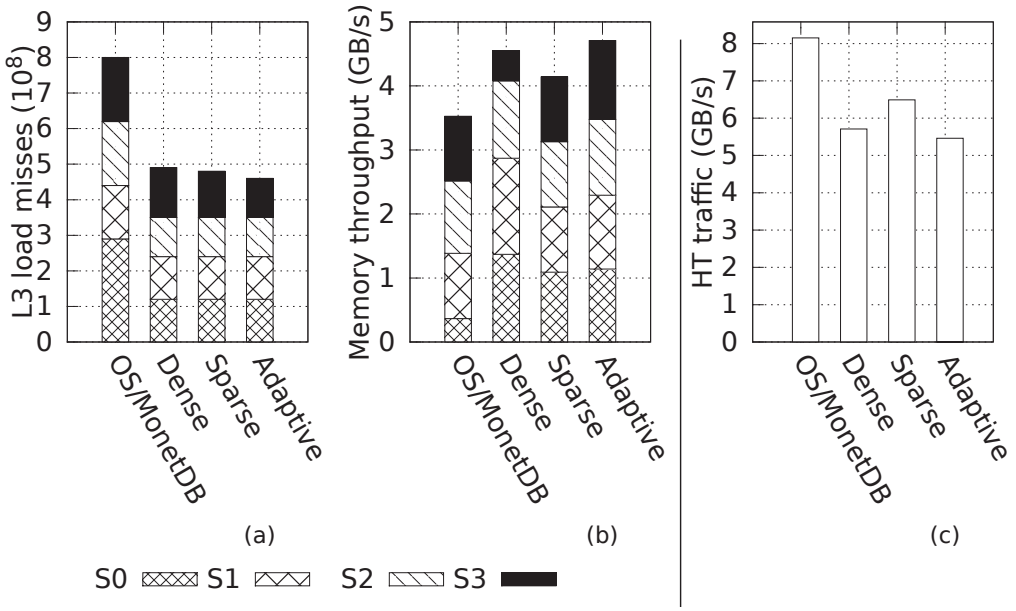


Figura 4.16: Métricas de acesso à memória com 256 usuários executando o *thetasubselect*.

e (b)). Com um mapeamento inferior, o escalonador do SO atingiu uma maior utilização de banda de interconexão (*HT Traffic*) e, consequentemente, um alto movimento de dados (Figuras 4.16 (c)). Com menos movimento de dados, o modo adaptativo explorou melhor a largura de banda de memória de todos os nós. Sendo que o modo denso permitiu a subutilização do nó 3 ($S3$), enquanto o modo esparso apresentou mais tráfego de interconexão e, consequentemente, menor *throughput* de memória. No entanto, ambos os modos apresentaram menos tarefas roubadas e melhor desempenho de memória do que o escalonador do SO.

4.5.2 Impacto da Seletividade

Esta seção apresenta o impacto dos diferentes modos de alocação na execução de 256 usuários concorrentes. Mediu-se o impacto das varreduras de colunas intensivas em memória, com diferentes seletividades [1]. A Figura 4.17 apresenta os resultados de erros de *cache* em L3 (*L3 cache miss*). Quanto menor o valor, melhor é a utilização de memória: observa-se essa situação em todos os modos de alocação do mecanismo proposto neste trabalho quando comparado ao escalonador do SO.

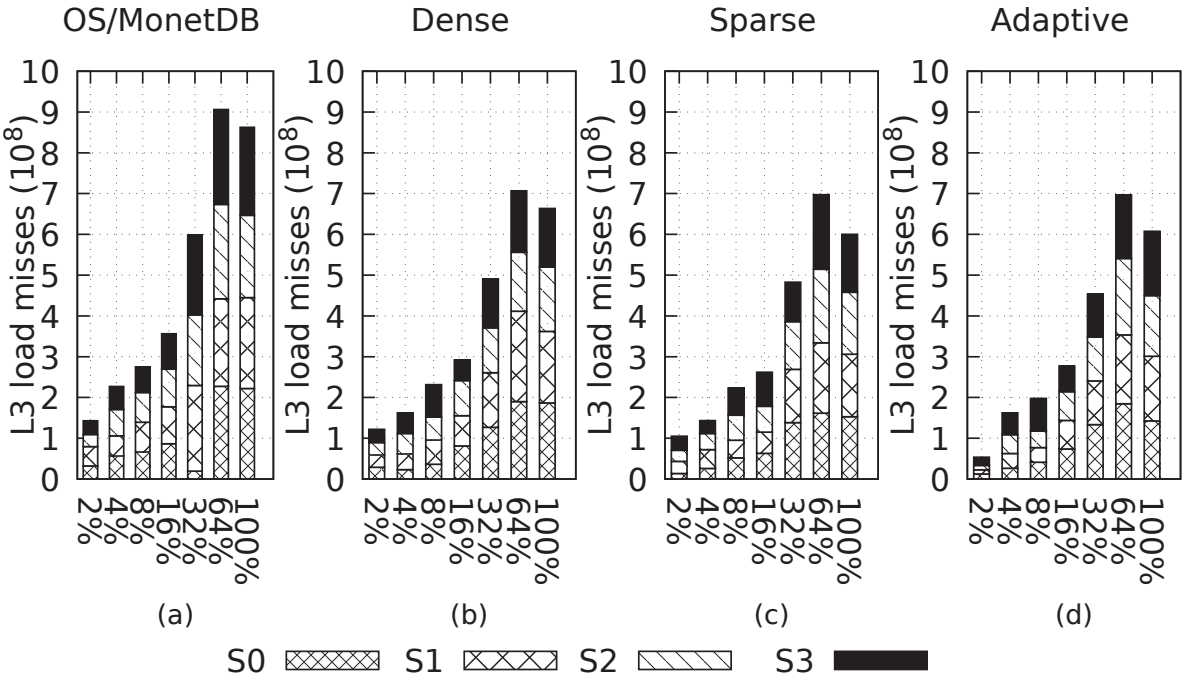


Figura 4.17: Avaliação de erros de *cache* em L3 com diferentes seletividades e com 256 usuários processando o *thetasubselect*.

A Figura 4.17 (a) mostra que o tamanho da *cache* é insuficiente para manter a materialização da consulta Q6 com mais de 64% de seletividade e com pico em *L3 load misses*. Nas Figuras 4.17 (b), (c) e (d), os modos de alocação do mecanismo compensam a insuficiência de *cache* com melhor rendimento de memória. Mesmo recuperando 100% da coluna, os modos de alocação utilizados pelo mecanismo de alocação de núcleo não apresentaram mais erros de *cache* em L3, comparado com o escalonador do SO (OS/MonetDB) com 64% de seletividade.

4.5.3 Impacto da Migração de *Threads*

A migração de *threads* é comparada durante a sua vida útil e analisado-se em quais núcleos as *threads* foram executadas. Nesta seção, é utilizado o plano de consulta completo da Q6 (ver Figura 4.14) para avaliar o impacto da migração de *threads* no acesso aos dados materializados em diferentes nós ao longo do *pipeline* de consulta. Os experimentos foram realizados em uma base de dados de 1 GB de tamanho.

Os modos denso e adaptativo fizeram o escalonador do SO executar as *threads*, na maioria das vezes, em um mesmo nó NUMA (Figura 4.18 (b) e (d)), enquanto o SO mapeava as *threads* do MonetDB por todos os nós NUMA (Figura 4.18 (a)). As *threads* migram várias

[1] A seletividade estima, em média, a porcentagem de registros na resposta de um determinado operador.

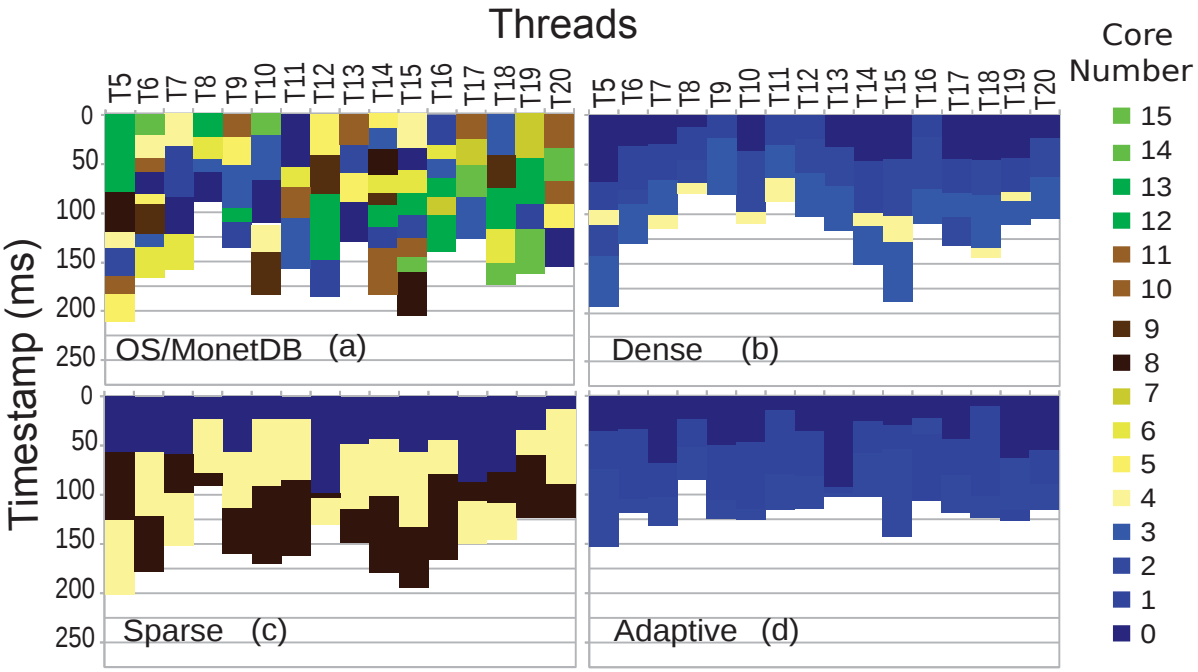


Figura 4.18: *Lifespan* e a migração de núcleos de cada *thread* do TPC-H Q6 original com um único cliente.

vezes de um núcleo para o outro, muitas vezes retornando ao mesmo núcleo. Isso mostra que o escalonador do SO não é consciente da arquitetura NUMA para a execução do banco de dados e constantemente está migrando as *threads* para encontrar afinidade NUMA e manter o balanceamento de carga. O modo esparso é o que mais se aproxima do comportamento esperado do escalonamento do SO, atribuindo as *threads* espalhadas nos nós NUMA. No entanto, a Figura 4.18 (c) mostra menos migração de *threads* nesse modo, quando comparado ao escalonador do SO, ao oferecer gradualmente menos núcleos ao SO.

São analisadas três métricas de desempenho quando o mecanismo expõe apenas o subconjunto ótimo de núcleos ao escalonador do SO. A Figura 4.19 (a) mostra que o tempo de resposta da consulta Q6 foi 27% mais rápido com o modo adaptativo comparado ao escalonador do SO. As Figuras 4.19 (b) e (d), respectivamente, mostram 9× mais tráfego de HT e 2× mais erros de *cache* em L3 para o escalonador do SO comparado ao modo adaptativo, impactando negativamente no desempenho e no consumo de energia. Com menos núcleos disponíveis para o mapeamento das *threads*, o mecanismo de alocação de núcleos auxiliou o SO a tomar boas decisões de escalonamento, resultando em mais acertos de *cache* e menos acessos remotos.

4.6 PETRINET: TRANSIÇÃO DE ESTADO COM HT/IMC

Esta seção apresenta os resultados do experimento em que é modificado o uso de CPU para a taxa de tráfego no controlador de memória integrado (HT/IMC). O objetivo do experimento é mostrar a flexibilidade do modelo abstrato, que pode se ajustar a diferentes estratégias e métricas para tomar decisões. A relação HT/IMC define como o sistema é consciente da arquitetura NUMA: o sistema pode processar mais dados com menos tráfego de interconexão. Desse modo, é possível observar se a utilização de interconexão alcança uma estratégia de alocação melhor que o uso de CPU. Os limiares foram definidos empiricamente na razão HT/IMC para $th_{min} = 0,1$ e $th_{max} = 0,4$, com base nos melhores resultados alcançados. Extraiu-se o HT/IMC de consultas dos seus TIDs na fila de prioridades.

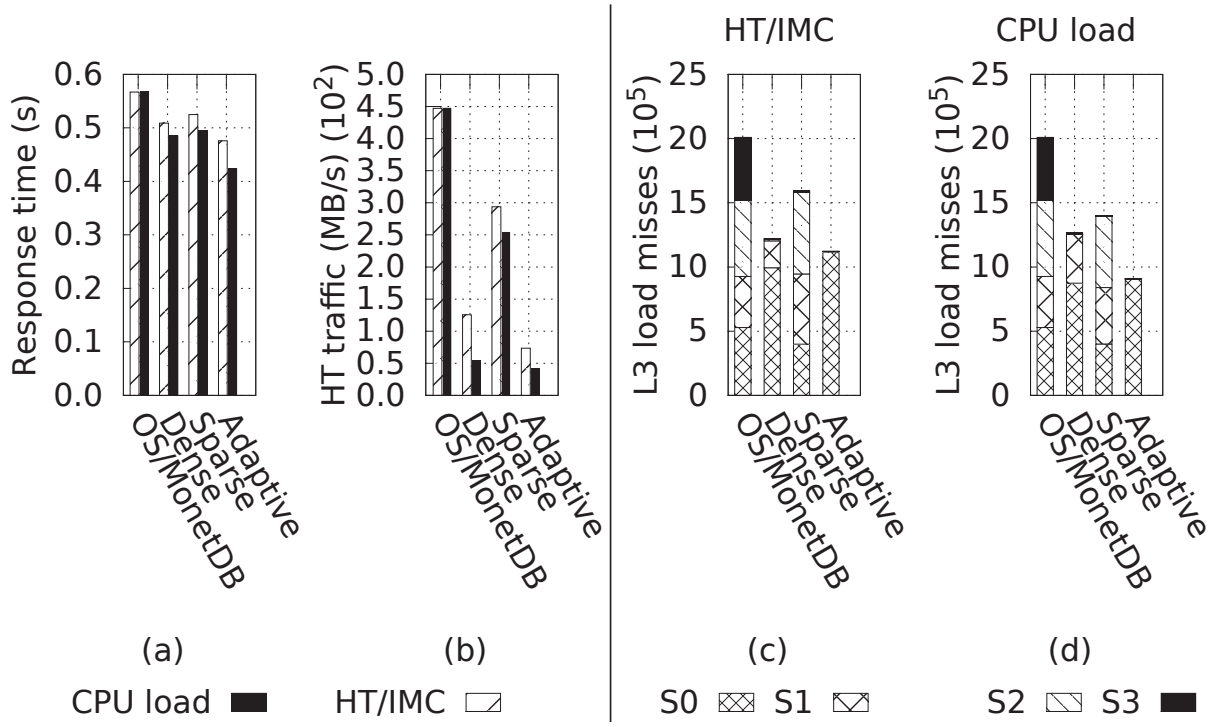


Figura 4.19: Métricas de desempenho da consulta Q6 com 1 usuário em um banco de dados de 1 GB executando nosso modelo com duas diferentes configurações de transição de estado: CPU *load* e HT/IMC.

A Figura 4.19 compara, lado a lado, a PrT configurada com ambas as estratégias de transição. De modo geral, o comportamento é similar aos resultados com pequenas diferenças no tempo de execução, tráfego de HT e erros em L3 (Figuras 4.19 (a), (b) e (c)). Em alguns casos, a carga de CPU apresentou metade do tráfego de HT. Com a métrica HT/IMC, o SO começou a preencher o *cache* L3. Mas, quando o mecanismo escolhe alocar um novo núcleo distante nessa métrica, todos os dados carregados em L3 são perdidos, gerando mais erros de *cache* e aumentando o tempo de execução (Figura 4.19 (c)). A transição de estado da PrT com HT/IMC demora mais tempo para fazer a alocação, com impacto nos movimentos de *threads*.

4.7 *BENCHMARK* TPC-H

Nessa seção, é avaliado o impacto do mecanismo de alocação de núcleos executando todas as consultas do *benchmark* TPC-H, usando duas cargas de trabalho. Além disso, o mecanismo é analisado com o SGBD relacional MonetDB e o *SQL Server* que é consciente da arquitetura NUMA. Tal execução não compara os dois SGBDs relacionais. Porém, o mecanismo de alocação de núcleo reage a um SGBD relacional não-consciente de NUMA e a um consciente de NUMA. O objetivo é mostrar que o mecanismo é ortogonal às estratégias de alocação de *threads* e dados, melhorando a afinidade de NUMA com uma alocação elástica de núcleos que auxilia o escalonador do SO.

4.7.1 Carga de Trabalho de Fases Estáveis

Os experimentos realizados nesta seção apresentam um cenário com um padrão de acesso mais dinâmico. Os resultados apresentados são de execuções simultâneas de todas as 22 consultas do TPC-H com objetivo de mostrar que o esquema de mapeamento de endereço proposto nesta tese autoadapta-se à carga de trabalho em mudança e mantém o processamento

em nós NUMA específicos. A carga de trabalho é dividida em fases. Em cada fase, é realizada a execução concorrente de cada consulta, uma por vez, por 256 usuários em um banco de dados de 1 GB. Na Figura 4.20, são apresentados a taxa de transferência de memória (eixo y) e o tempo de execução (eixo x).

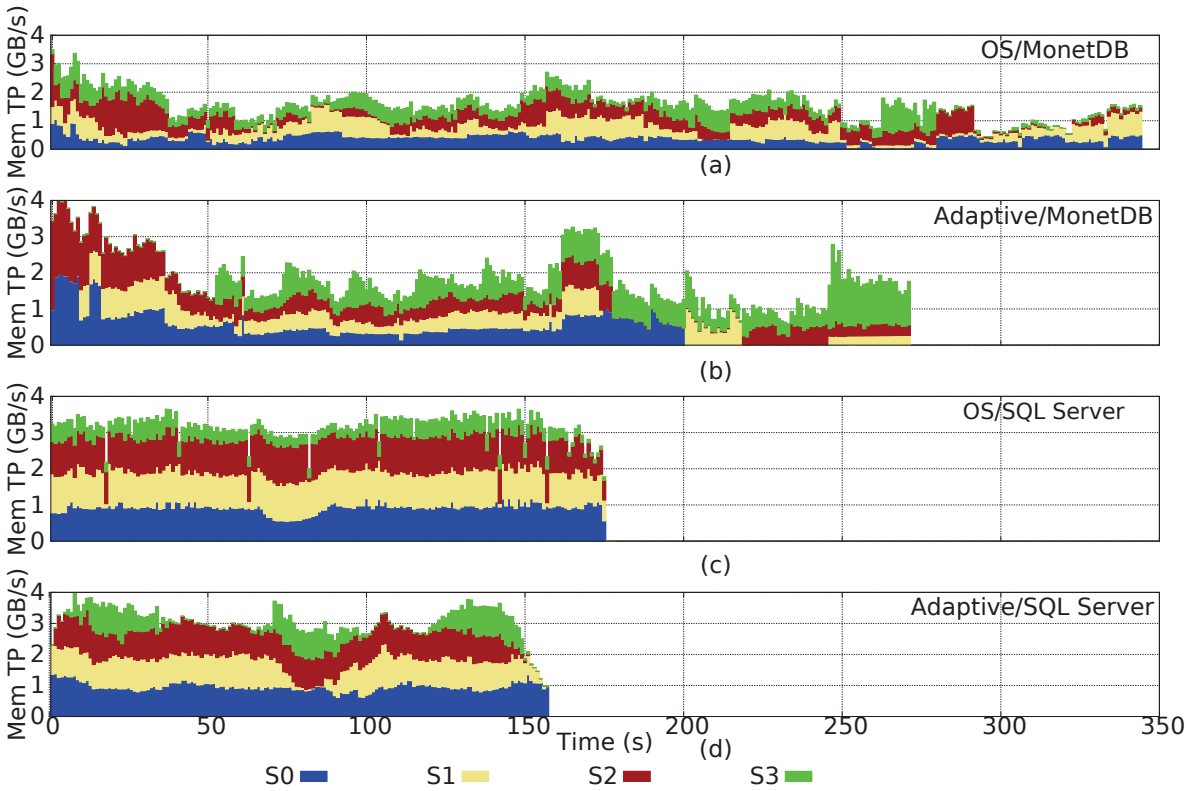


Figura 4.20: Carga de trabalho de fases estáveis com 256 usuários concorrentes em um banco de dados de 1 GB.

No SGBD relacional MonetDB, o modo adaptativo melhorou a tarefa de mapeamento do SO para encontrar afinidade de dados e *threads*, sendo 41% mais rápido com maior taxa de transferência de memória do que o OS/MonetDB. Por exemplo, a execução em 50-ésimo segundo mostra que o mecanismo se concentrou nos nós $S0$ e $S2$. Em outro caso, entre 200-ésimo e 220-ésimo segundo, o mecanismo aqui apresentado se concentrou nos nós $S1$ e $S3$. Além disso, observa-se que, sem o mecanismo proposto nesta tese, o SO ignora a carga do banco de dados e usa constantemente os núcleos no soquete $S0$ durante toda a execução (Figuras 4.20 (a) e (b)).

Ao executar com o *SQL Server*, é possível observar que um SGBD consciente da arquitetura NUMA explora melhor os nós simultaneamente. Porém, mesmo assim, o mecanismo de alocação de núcleos diminuí o tempo de resposta. O *SQL Server* associa as *threads* e núcleos para melhorar a afinidade, mas com menos núcleos disponíveis, por exemplo, entre o 40-ésimo e o 60-ésimo segundo, há menos esforço para manter a coerência de tal associação (Figuras 4.20 (c) e (d)).

4.7.2 Carga de Trabalho de Fases Mistas

Os experimentos foram executados com 256 usuários simultâneos executando continuamente uma consulta aleatória das 22 consultas do TPC-H. Nas Figuras 4.21 e 4.22, são apresentados os resultados de *speedup* de tempo de resposta por consulta do mecanismo proposto comparado ao escalonador do SO com o MonetDB e *SQL Server* respectivamente, assim como a proporção de HT/IMC descrita em [Porobic et al. 2012].

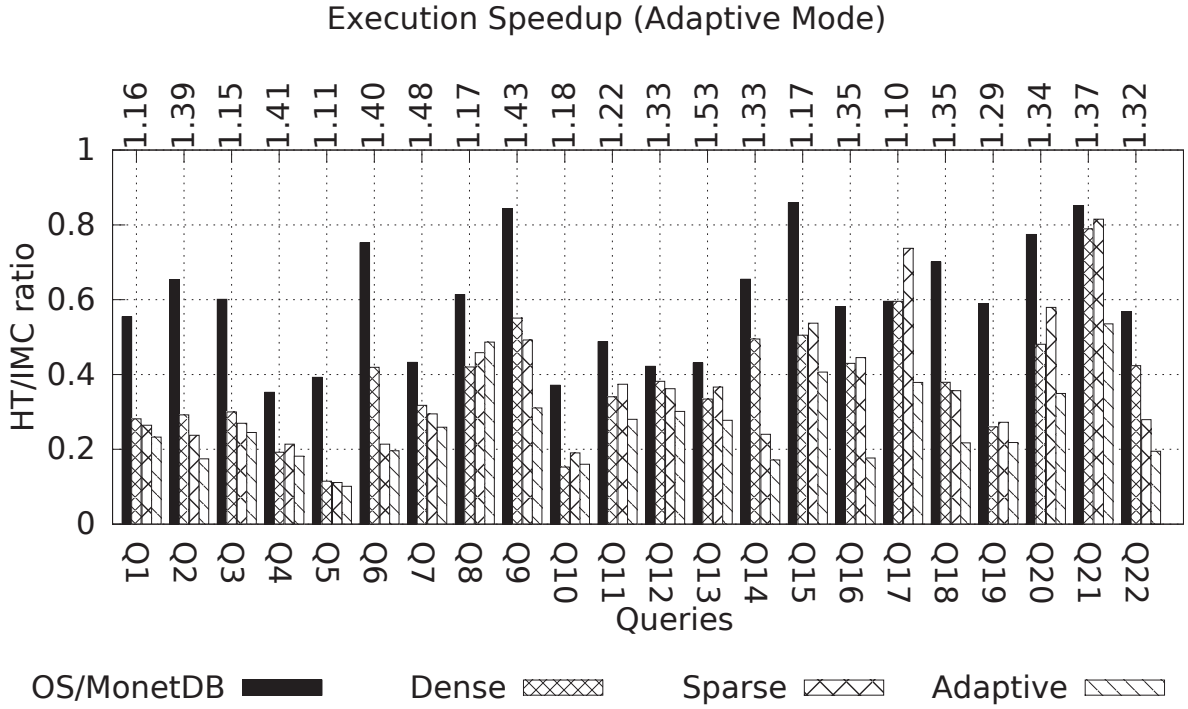


Figura 4.21: Execução da carga de trabalho de fases mistas por consulta com 256 usuários simultâneos em uma base de dados de 1 GB no MonetDB. No eixo *Y*, a relação HT/IMC mostra quão consciente da arquitetura NUMA é o sistema (quanto menor melhor). No topo, é o aumento de desempenho (*speedup*) para o modo adaptativo.

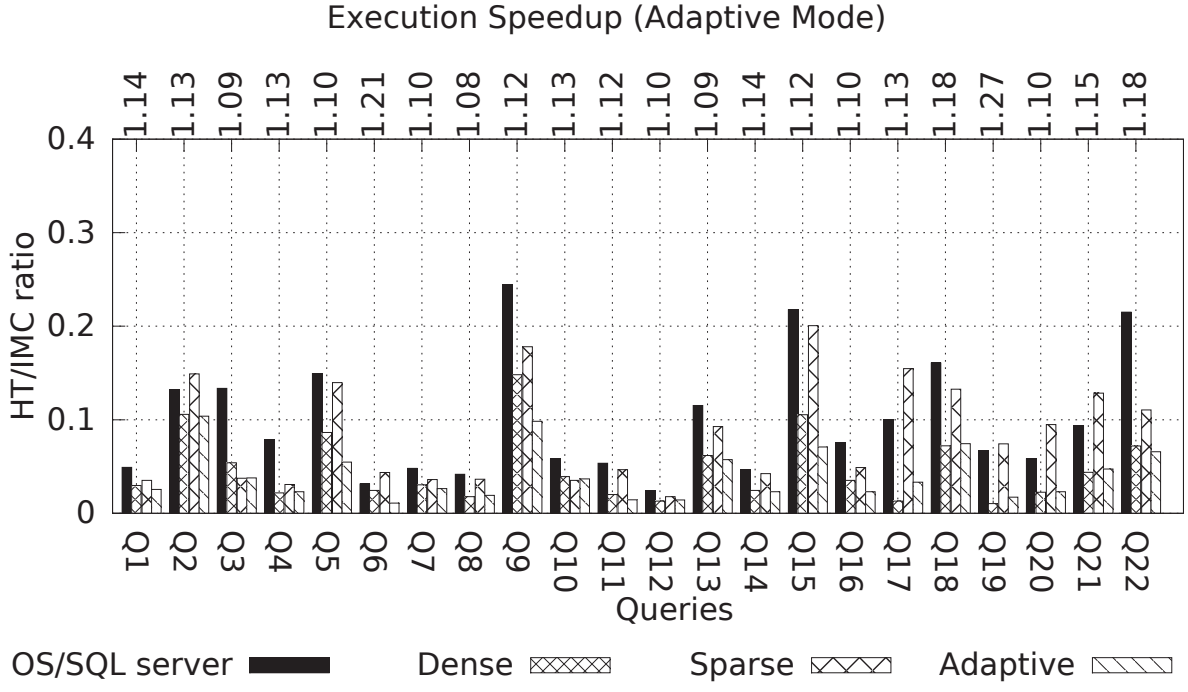


Figura 4.22: Execução da carga de trabalho de fases mistas por consulta com 256 usuários simultâneos em uma base de dados de 1 GB no *SQL Server*. No eixo *Y*, a relação HT/IMC mostra quão consciente da arquitetura NUMA é o sistema (quanto menor melhor). No topo, é o aumento de desempenho para o modo adaptativo.

O modo adaptativo apresentou o melhor desempenho por consulta, alcançando até $1,48\times$ e até $4\times$ menos HT/IMC se comparado ao escalonador do SO com o MonetDB. No *SQL Server*, o *speedup* foi de $1,27\times$, atingindo HT/IMC $4\times$ menor. No MonetDB, observa-se $3\times$

menos taxa de HT/IMC para as consultas Q8 e Q9 (*SQL Server* 2× menor). Essas consultas possuem o maior número de operações de junção, com um alto grau de paralelismo. As *threads* das consultas Q8 e Q9 encontram mais dados na memória local em comparação com o SGBDs relacionais sem o modo adaptativo. As consultas Q19 e Q22 apresentaram 3× menos HT/IMC no MonetDB e no *SQL Server* porque o mecanismo de alocação de núcleo reduziu o número de núcleos em que as *threads* processam predicados ”IN“ (uma série de valores constantes compartilhados em uma lista). A diminuição no uso de interconexão é um reflexo direto do modo adaptativo: núcleos alocados em nós com mais páginas acessadas e núcleos liberados com o menor número de páginas acessadas. Portanto, as *threads* exigem menos acessos remotos e migrações, aumentando a eficiência para encontrar afinidade de dados.

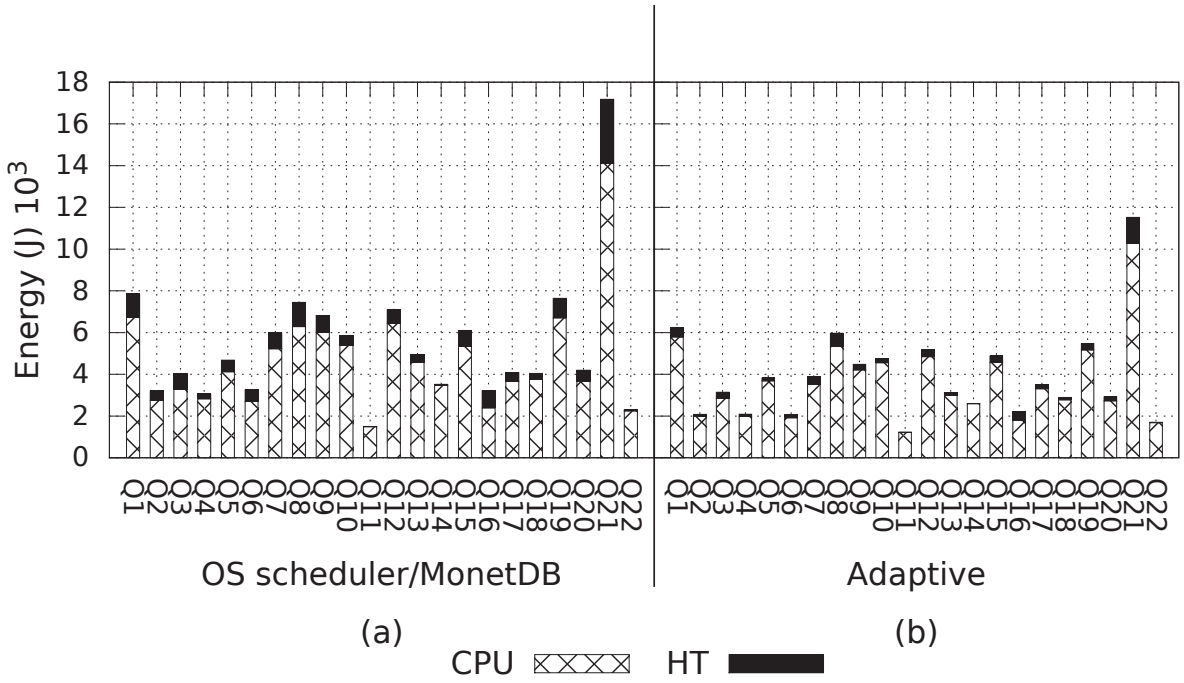


Figura 4.23: Estimativas de energia para CPU e HT, executando as consultas em um banco de dados de 1 GB TPC-H com 256 clientes simultâneos no MonetDB.

4.7.3 Avaliação de Energia

Esta seção avalia o efeito do mecanismo de alocação de núcleo no consumo de energia. As estimativas foram feitas baseadas no melhor modo de alocação, o adaptativo, comparado ao escalonador do SO com o MonetDB e *SQL Server*. Foram utilizados os valores referentes à potência média da CPU (ACP) para o processador usado nos experimentos. Também foi obtida a energia média por *bit* transferida para HT [Wang e Lee 2016]. As estimativas foram realizadas para todo o *benchmark* executado em experimentos anteriores, utilizando os resultados de contadores de *hardware*.

As Figuras 4.23 e 4.24 mostram o consumo de energia em Joules durante a execução das 22 consultas (dividida entre CPU e HT) nos SGBDs relacionais MonetDB e *SQL Server*. A maior parte da economia de energia da CPU veio do menor tempo de execução, enquanto a economia de energia do HT veio do número reduzido de transferências de dados. Para o MonetDB, foram obtidas economia mínima de $9,67\%$ para a CPU (consulta Q17) e HT de $46,22\%$ (consulta Q12). Analisando por média geométrica, o mecanismo proposto economizou

22, 93% para a CPU e 63, 20% para HT, com uma economia total de 26, 05% no consumo do sistema. No SGBD relacional *SQL Server*, a economia mínima no consumo de energia para a CPU foi de 8, 14% (consulta Q8) e de 13, 8% para a HT (consulta Q13), com uma economia total de 12, 03% para CPU e 49% para HT. O mecanismo de alocação de núcleo melhorou o escalonamento das *threads* do banco de dados com redução direta nas transferências de dados entre os nós NUMA, levando a um melhor consumo de energia.

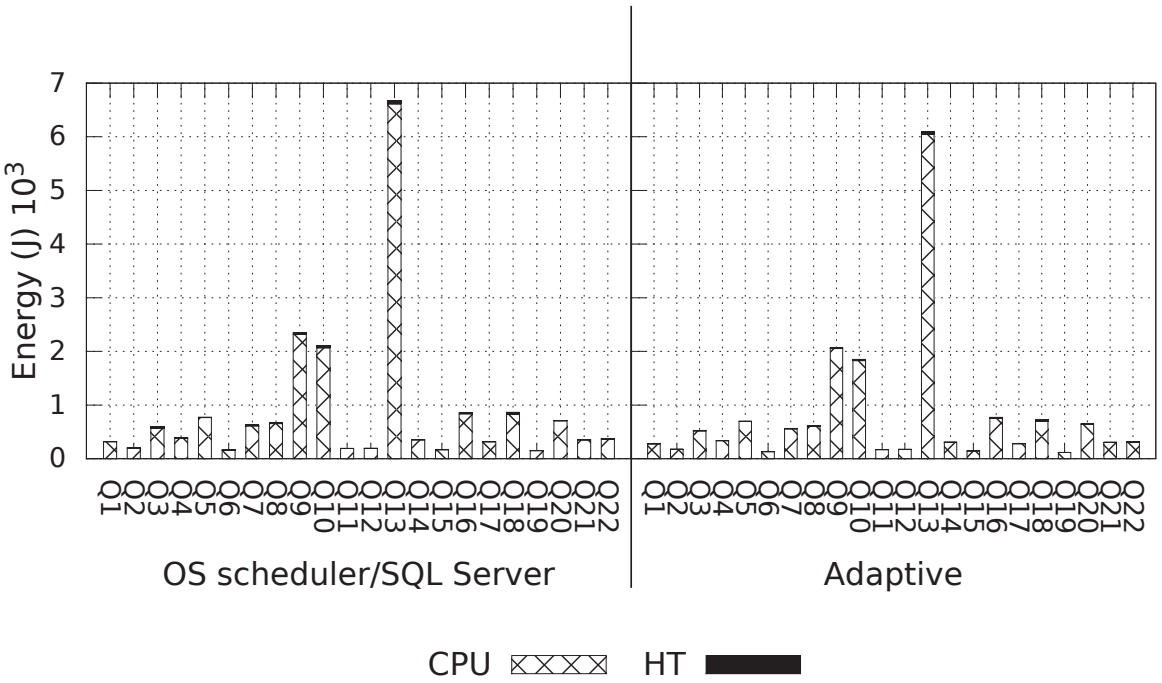


Figura 4.24: Estimativas de energia para CPU e HT, executando as consultas em um banco de dados de 1 GB TPC-H com 256 clientes simultâneos no *SQL Server*.

4.7.4 Impacto do escalonamento com limiares dinâmicos

Essa seção apresenta a análise dos efeitos dos limiares dinâmicos executando o modo de alocação adaptativo, à medida que aumenta a concorrência, com o número de usuários em paralelo. Os experimentos foram variados até 256 usuários, executando a versão modificada da consulta Q6 em 1 GB, analisando-se as métricas de desempenho como pode ser observado na Figura 4.25. Os resultados não apresentaram muitas variações comparado ao valor fixo de limiar superior em 70% e limite inferior de 10% adotados. De modo geral, os resultados alcançados com o mecanismo foram conservados, como era esperado, pois, as variações nos limiares dinâmicos foram pequenas.

Com pouca variação do limiar dinâmico e mantendo os resultados obtidos, o mecanismo torna-se completamente dinâmico, sem necessidade de interferência de um administrador do sistema. Nos experimentos realizados de uso de CPU, os limiares foram ajustados com base na carga de uso de recursos ao executar o mecanismo de alocação e na quantidade de núcleos já alocados. A variação dos limiares ocorre quando a quantidade de núcleos varia. Os valores dessa variação são mantidos ao longo da execução. Por exemplo, na arquitetura com 16 núcleos, encontram-se 16 limiares diferentes enquanto o SGBD relacional está estável ou sob controle. Na sequência, os limiares são utilizados para detectar possíveis alterações quando o número ótimo de núcleos corresponde ao calculado.

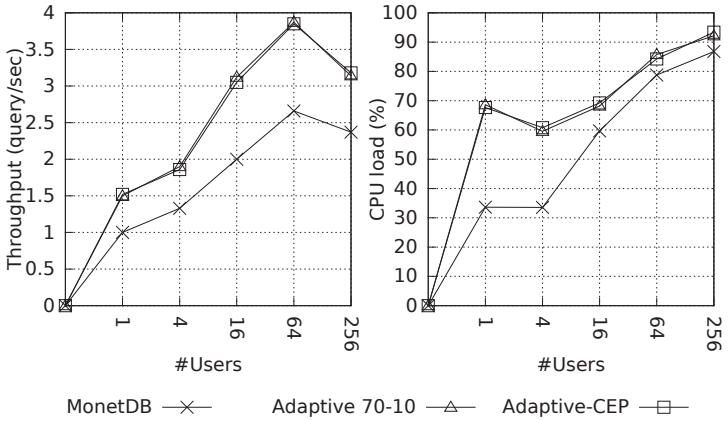


Figura 4.25: Métricas de desempenho ao processar um número crescente de usuários simultâneos executando o *thetasubselect* com limiares dinâmicos.

4.8 CONSIDERAÇÕES FINAIS

O mecanismo de alocação de núcleo apresentado neste capítulo mitigou o movimento de dados e conseguiu melhorar o desempenho da execução da carga de trabalho analíticas. Assim, os resultados apresentados mostram melhorias de desempenho quando o mecanismo ofereceu ao SO apenas os núcleos de processamento necessários para a execução, comparado a abordagem tradicional de deixar todos os núcleos disponíveis.

O mecanismo adaptativo melhorou a precisão do escalonamento do SO quando os núcleos foram disponibilizados gradualmente. Obtiveram-se menos migrações de *threads* e menos acessos à memória remota em comparação ao tradicional escalonador do SO. Utilizando as informações de recursos computacionais internos do SO e do SGBD relacional, foi explorado o potencial de implementação de algoritmos de alocação no nível abstrato.

Portanto, observa-se um importante *speedup* de até $1,53\times$ com redução de até $3,87\times$ na relação HT/IMC por consulta quando comparado ao escalonador do SO tradicional. As estimativas mostram que essa redução no número de acessos remotos leva a uma economia total de energia de $26,05\%$ no sistema NUMA. Ao avaliar o mecanismo de alocação de núcleo com o *SQL Server* compatível com NUMA, foi possível observar um *speedup* de até $1,27\times$ e redução de até $3,70\times$ por relação de HT/IMC de consulta. Isso destaca a necessidade do modelo de alocação de núcleo adaptativo para melhorar a afinidade NUMA.

5 ANÁLISE DE DESEMPENHO DE SISTEMAS DE BANCO DE DADOS DE MATRIZ EM ARQUITETURA DE MEMÓRIA NÃO-UNIFORME

Os SGBDs relacionais são amplamente utilizados em muitas aplicações devido ao esquema eficiente de armazenamento proporcionado pelo modelo relacional, que permite reduzir o espaço de armazenamento e facilitar a manutenção dos dados. Esse modelo também permite que os usuários combinem e busquem dados de maneira flexível, usando uma linguagem de consulta declarativa. No entanto, o modelo relacional não é ideal para armazenar e analisar dados científicos devido às suas limitações e incompatibilidade para trabalhar diretamente com o formato dos dados gerados em simulações e experimentos científicos: a matriz de dados [Blanas et al. 2014; Lustosa e Porto 2019]. A incompatibilidade entre a matriz e o modelo relacional requer uma série de transformações de dados para executar consultas de modo a analisar dados científicos [Brown 2010].

Os SGBDs multidimensionais implementam modelos de dados multidimensionais para servir a muitas aplicações que geram muitos dados que não se encaixam no modelo relacional tradicional, como simulações científicas, exploração de dados, aplicações que envolvem aprendizado de máquina, simulação ou análise de estruturas biológicas, para citar apenas alguns [Baumann e Holsten 2011; Kim et al. 2016]. Normalmente, essas áreas produzem rapidamente grandes volumes de dados binários para serem processados e analisados de uma só vez. As linguagens de consulta de matriz fornecem operações multidimensionais específicas, como operações de álgebra linear e geométrica (por exemplo, fatias de dados, transposição de matriz, adição, subtração, matriz oposta), e usam paralelismo para reduzir o tempo de resposta da consulta.

Ao mesmo tempo, os sistemas de alto desempenho usam arquiteturas NUMA para aproveitar o alto número de núcleos de CPU em uma mesma memória compartilhada. Na arquitetura NUMA, a latência de acesso à memória pode variar dependendo do endereço que está sendo acessado – por exemplo, acesso a nós de computação próximos ou distantes. Nesse cenário, os bancos de dados SGBDs relacionais e SGBDs multidimensionais podem apresentar desempenhos diferentes, dependendo de como as *threads* de consulta são posicionadas e de como o mapeamento de dados ocorre nos nós.

Os modelos de processamento de consulta usados por SGBDs multidimensionais são semelhantes aos usados por SGBD relacional. Por exemplo, o SGBD SAVIME [Lustosa et al. 2017] usa o modelo de consulta de materialização, enquanto o SGBD SciDB [Brown 2010] usa o modelo de consulta de *pipeline*. Além disso, por padrão, os SGBDs multidimensionais contam com o SO para mapear *threads* de consulta para núcleos de CPU, conforme observado no SGBD relacional tradicional. Eventualmente, pode-se perguntar se o impacto negativo observado no SGBD relacional descrito no capítulo 4 se mantém quando a arquitetura NUMA é usada para oferecer suporte a outros modelos de sistemas de banco de dados.

Neste capítulo, é avaliado o desempenho SGBDs multidimensionais em execução em uma arquitetura NUMA. Para isso, são utilizadas várias estratégias de alocação de *thread* estáticas conhecidas com o objetivo de medir possíveis melhorias no desempenho da consulta. Esta pesquisa é a primeira a avaliar esse impacto nos SGBDs multidimensionais. Nos experimentos, foram adotados os sistemas de última geração SAVIME e SciDB que implementam um modelo de dados de matriz multidimensional do zero, ou seja, sem adaptações ao modelo relacional. As principais contribuições apresentadas neste capítulo são:

- **Comparação de técnicas tradicionais:** são analisados a aceleração e o impacto da energia de cinco estratégias diferentes de alocação de *threads* para sistemas NUMA ao executar o SAVIME e o SciDB.
- **Análise de desempenho máximo:** é mostrado que as técnicas tradicionais de distribuição de *threads* entre os núcleos NUMA ainda estão longe de ser um ponto ótimo de melhoria.

Este capítulo está organizado da seguinte forma: a próxima seção, 5.1, descreve um pouco sobre os SGBDs multidimensionais; a seção 5.3 mostra os resultados empíricos; e a seção 5.4 traz as considerações finais.

5.1 SISTEMAS DE BANCO DE DADOS DE MATRIZ

Nesta seção, são descritos brevemente o modelo de dados dos SGBDs multidimensionais e o operador de consulta para dividir os dados multidimensionais que foi utilizado nas avaliações e experimentos. Com o conceito de modelo de dados de matriz, uma ampla gama de SGBDs multidimensionais surgiu, como RasdMan [Baumann et al. 1997], ArrayStore [Soroush et al. 2011], SciDB [Brown 2010], SciQL [Zhang et al. 2013] e SAVIME [Lustosa et al. 2017]. Como citado, este estudo concentra-se em dois bancos de dados *full-stack*, SAVIME e SciDB, que implementam o modelo de matriz do zero, sem fazer adaptações no modelo relacional. Em máquinas de multiprocessamento escalonáveis, isso permite a distribuição de blocos de dados em nós de máquina separados.

Os *chunks* são uma representação física de uma matriz e ambos os sistemas dividem as matrizes em blocos conforme os tipos de dados armazenados. O tamanho e o formato do bloco dependem da densidade da matriz. Para matrizes densas, todos os *chunks* terão o mesmo tamanho. Por outro lado, quando as matrizes são esparsas, os *chunks* podem ter tamanhos e formatos diferentes. Os *chunks* não-regulares – ou seja, matrizes esparsas – são propensos a serem distribuídos de maneira não-uniforme, causando penalidades no processamento de consultas. As consultas em um SGBD multidimensional extraem dados de matrizes multidimensionais usando chamadas de funções aninhadas. Os *chunks* são canalizados por meio de operadores de consulta. A estrutura de dados de matriz usa índices para tornar o acesso a um conjunto de células mais rápido. Além disso, se uma consulta frequentemente usa um trecho para acesso mais rápido, o trecho é mantido na memória [Gerhardt et al. 2015].

Os bancos de dados SAVIME e SciDB suportam uma linguagem funcional de matriz (AFL) com uma série de operadores definidos como funções [Lustosa e Porto 2019; Kim et al. 2016]. Neste capítulo, o foco é no operador classificado pelo SciDB como operador de seleção. A mesma operação possui nomes diferentes em SGBDs multidimensionais: no SAVIME, é *subset*; e, no SciDB, é *subarray*. A partir deste ponto, este trabalho se referirá a esta operação como *subarray*. O operador de *subarray* usa índices de dimensão para acesso rápido aos dados no SAVIME, selecionando os dados em um intervalo e projetando os mesmos. Já no SciDB utiliza os índices para criar uma cópia dos dados dentro de um intervalo.

Listing 5.1: Filtrando a matriz (representada na Figura 2.7) de temperatura 3D usando a API de *subarray* SciDB.

```
1 subarray ( temperatura ,−25.423,−49.267,2000,−25.426,−49.265,2020);
```

Os SGBDs multidimensionais implementam operadores de *subarray* de maneiras diferentes. O SAVIME encontra células entre o intervalo usando o filtro e gera muitos *chunks* com tamanhos diferentes como resultado. O SciDB decodifica os dados binários compactados para os *chunks* e redistribui os dados para produzir uma nova configuração de *chunks* para os resultados

que estão na faixa de interesse. O *subarray* é uma operação simples em um modelo de dados de matriz. No entanto, conforme a seletividade da consulta, pode-se exigir o processamento de todos os *chunks*. Na Figura 5.1 está representando dois exemplos da operação de *subarray*. Os dois exemplos selecionam dados de ambos os *chunks* da matriz que estão representados pelas cores claras. A operação de *subarray* é representada pela cor azul e projeta parte da matriz total.

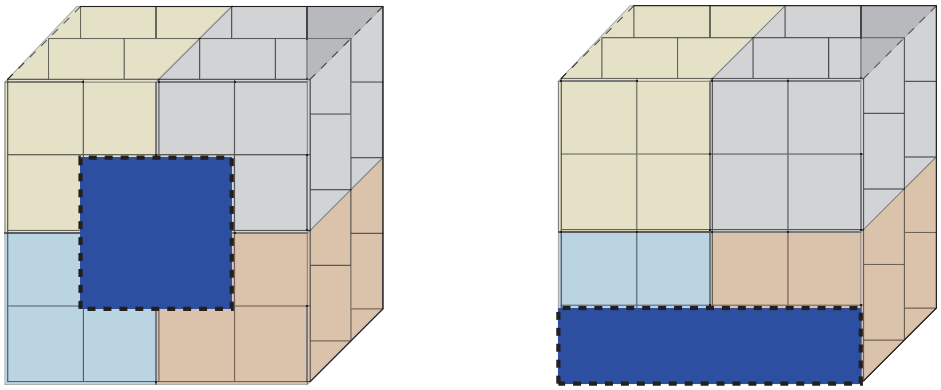


Figura 5.1: Dois exemplos da operação de *subarray* em uma matriz multidimensional

5.2 ESTRATÉGIAS DE ALOCAÇÃO DE *THREADS*

Nesta subseção, são descritas cinco estratégias de alocação de *thread* bem conhecidas usadas nos experimentos. A estratégia *sparse* é semelhante com a utilizada na seção 4.1 no capítulo 4. O objetivo é analisar o impacto dessas estratégias em comparação com o *baseline*, ou seja, em comparação com a estratégia de agendamento do SO. Foi utilizada, para o Savime, a afinidade de *thread* disponível na linguagem *OpenMP* [Committee 2013]. Para o SciDB, foi utilizado o *taskset*[1] para empregar a afinidade das *threads* nos núcleos NUMA. Tanto o *baseline* quanto as cinco estratégias avaliadas estão descritas a seguir:

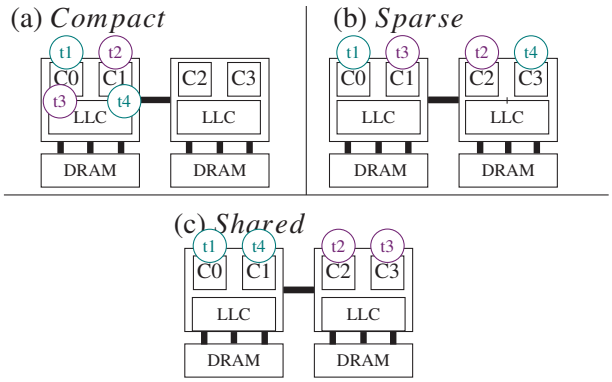


Figura 5.2: Estratégias de alocação de *threads Compact*, *Sparse* e *shared* em dois nós NUMA com 4 *threads*.

- ***Baseline*:** para fornecer um *baseline* comumente usado, foi decidido medir o desempenho de alocação de *thread* do SO sem a interferência do usuário deixando o mapeamento para o SO. O SO usa um balanceamento de carga para distribuir *threads* em todos os nós NUMA com o objetivo de maximizar o uso dos núcleos.

[1] `https://man7.org/linux/man-pages/man1/taskset.1.html`

- **Estratégia 1 — *Compact*:** na Figura 5.2 (a), pode ser observada a estratégia *compact*. A alocação de *thread compact* segue a ordem de criação de *threads* e usa apenas um nó NUMA. Essa estratégia pode fornecer benefícios de carga de trabalho com grande reutilização de dados. No entanto, o tamanho dos dados é limitado à memória disponível no nó.

- **Estratégia 2 — *Sparse*:** a estratégia *sparse* distribui as *threads* igualmente entre os nós, um encadeamento por nó. Por exemplo, $t1$ é alocado no nó 0, $t2$ no nó 1 e assim por diante, como mostrado na Figura 5.2 (b). Portanto, o objetivo é medir o desempenho ao espalhar as *threads* nos nós e fixá-los em um núcleo.

- **Estratégia 3 — *Shared*:** a estratégia *shared* visa fixar conjuntos de *threads* que funcionam no mesmo *chunk* a um único nó NUMA. Essas *threads* compartilham o nó do último nível de *cache* (LLC), conforme mostrado na Figura 5.2 (c). Essa estratégia foi utilizada para analisar se a reutilização de dados tem um impacto positivo na minimização dos efeitos da arquitetura NUMA.

- **Estratégia 4 — *Petrinet*:** nessa estratégia, é utilizado o mesmo modelo apresentado no capítulo 4. O mecanismo dinâmico implementa um modelo abstrato baseado na rede de Petri. O objetivo desse mecanismo é manter um número ideal local de núcleos para lidar com a carga de trabalho do banco de dados atual. A suposição é que um número mínimo de núcleos mantém o desempenho. A principal métrica para atingir o número ideal de núcleos é a carga da CPU.

- **Estratégia 5 — *Random*:** aqui, as *threads* do SGBD multidimensional são posicionadas aleatoriamente nos núcleos em todos os nós NUMA. Foram geradas 20 alocações aleatórias para verificar se as cargas de trabalho têm comportamentos diferentes. O objetivo é evitar a alocação de núcleo idêntica, como nas estratégias *compact* e *sparse*, quando as *threads* são posicionados nos mesmos núcleos correspondentes. A estratégia *random* identifica se existe uma alocação de *thread* que melhora o desempenho em comparação com as outras estratégias avaliadas.

5.3 ANÁLISE EXPERIMENTAL

Nesta seção, são avaliados os desempenhos do SGBDs multidimensionais SAVIME na versão (v.1.0) e do SciDB na versão (v.19.11.5), em duas máquinas NUMA, usando as estratégias de alocação de *threads* descritas anteriormente. Basicamente, a primeira máquina é utilizada nos experimentos e, quando apropriado, é apontado o uso da segunda máquina.

A primeira máquina NUMA – aqui chamada de *NUMA-Skylake* – tem dois nós, cada nó com um *Intel Xeon Silver* 4114 (com microarquitetura *Skylake*). Cada soquete *Xeon* tem dez núcleos com *cache* L1 (I + D) privado (32 KB cada núcleo), um *cache* L2 privado (1 MB cada núcleo) e um *cache* L3 compartilhado (total de 14 MB por nó). Os dois nós NUMA são interconectados por um *link* QPI [Intel 2019] 4×, com largura de banda de 21, 5 GB/S. A máquina inclui 128 GB de memória principal DDR-4 e 14 TB de armazenamento em disco (a 15.000 rpm), executando o SO Ubuntu na versão 18.04.01 LTS para SAVIME e na versão 14.04.6 LTS para SciDB, que no momento do desenvolvimento desta tese executava somente nesta versão. As diferentes versões do SO foram usadas conforme a documentação do SGBD multidimensional, em um *kernel* Linux não-modificado, versão 4.15.0-121, genérico.

A segunda máquina – aqui chamada de *NUMA-SandyBridge* – também possui dois nós, cada nó com um *Intel Xeon E5* — 2630, com microarquitetura *Sandy Bridge*. Cada nó

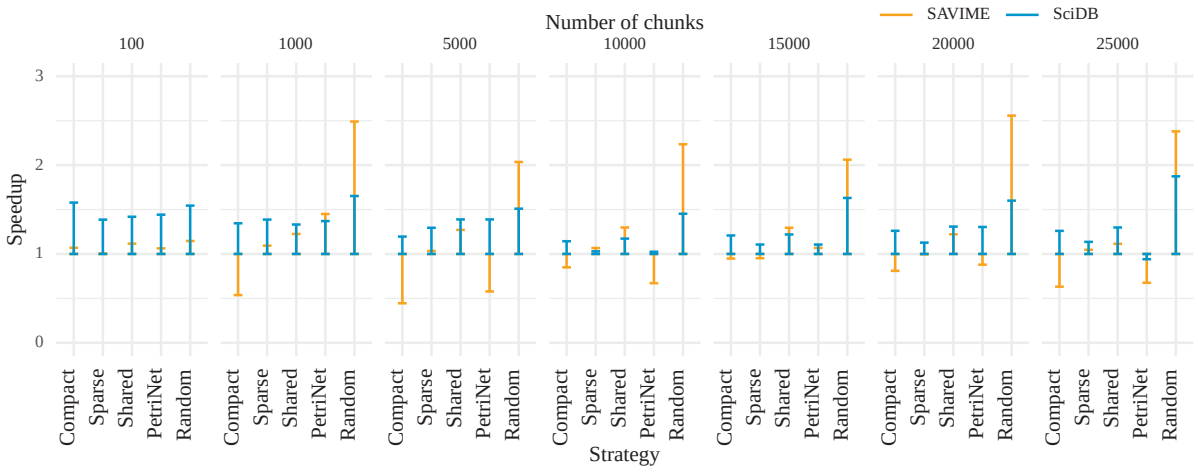


Figura 5.3: Comparação do desempenho do operador *subarray* em um banco de dados de 1 GB, usando diferentes números de *chunks*, no SGBD multidimensional na máquina *NUMA-Skylake*.

Xeon tem seis núcleos físicos com *cache* L1 (I + D) privado (32 KB cada núcleo), um *cache* L2 privado (256 KB cada núcleo) e um *cache* L3 compartilhado (total de 15 MB por nó). Os dois nós NUMA são interconectados por um *link* QPI 4×, com largura de banda de 14, 4 GB/S. A máquina inclui 48 *GB* de memória principal DDR-3 e 869 GB de armazenamento em disco (a 7500 rpm), executando o SO CentOs na versão 7.9.2009.

Para medir o desempenho do *hardware*, utilizou-se o *Intel Performance Counter Monitor* (PCM) [Willhalm Thomas 2012]. Em particular, a ferramenta *Intel* PCM fornece o consumo total de energia da memória principal. Para gerenciar as estratégias de alocação de *thread*, foram utilizados o *OpenMP* (versão 4.5) *thread affinity* [Committee 2013] e o comando *taskset* Linux. Nos experimentos, definiu-se que o número máximo de *threads* disponíveis para cada execução de consulta corresponderia ao número de núcleos físicos disponíveis. A carga de trabalho possui uma matriz densa baseada em dados do *benchmark* sísmico *HPC4e BSC* [Center 2016], usado no trabalho apresentado em [Lustosa e Porto 2019]. Os resultados são apresentados considerando a média de mais de 10 execuções para cada estratégia de alocação de *thread* e *baseline*.

A análise é concentrada no operador de *subarray*. Esse operador foi escolhido devido ao seu comportamento simples de acesso à memória e à sua quantidade considerável de aplicações [Lustosa e Porto 2019; Papadopoulos et al. 2016]. Essa operação tem um padrão de acesso à memória coalescente devido ao predicado de desigualdade que recupera intervalos de células da matriz. Não há reutilização de dados de *cache* na operação de *subarray* devido ao seu comportamento de *streaming* de dados. Além disso, vários processamentos paralelos dessa operação de *subarray* levam à movimentação de dados entre os nós NUMA, com impacto direto no desempenho dos SGBDs multidimensionais.

5.3.1 Impacto do número de *chunks*

Nesta seção, é investigado o impacto do número de *chunks* em uma matriz de 1 GB. É necessário observar que um *chunk* é a menor unidade de armazenamento em um SGBD multidimensional. Para esta configuração, sempre que se aumenta o número de *chunks* para uma determinada matriz, o tamanho de cada *chunks* diminui, mantendo assim o mesmo tamanho total de armazenamento. Neste experimento, todas as configurações e estratégias executam a mesma operação no mesmo conjunto de dados, gerando a mesma saída. Além disso, a execução dos experimentos foi na máquina *NUMA-Skylake*.

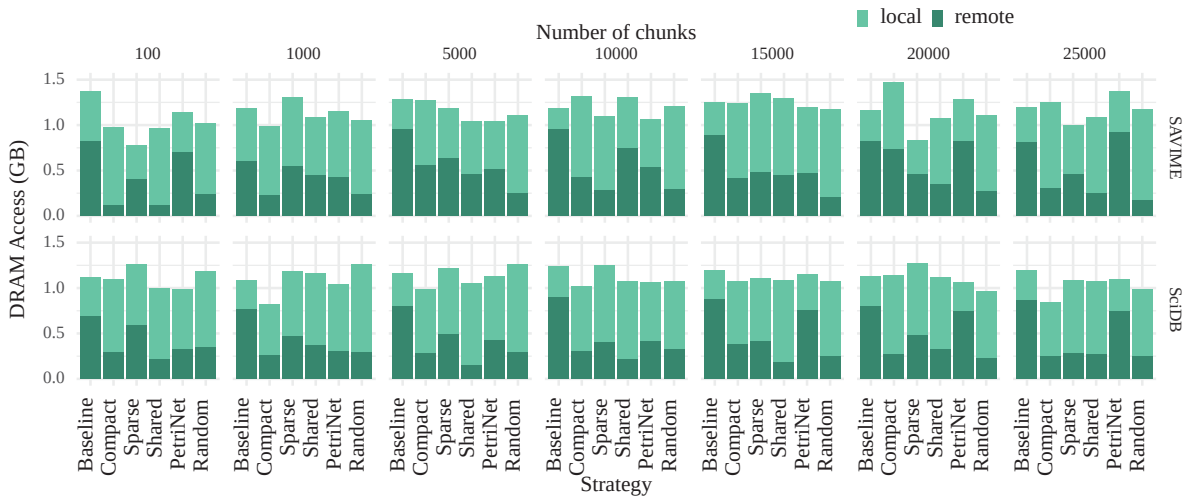


Figura 5.4: Acessos à memória remotos e locais na operação de *subarray*, em um banco de dados de 1 GB, usando diferentes números de *chunks* nos SGBDs multidimensionais.

A Figura 5.3 apresenta os resultados para o SAVIME e para o SciDB, em *speedup* por quantidade de *chunks* ao usar as diferentes estratégias de alocação de *thread* em comparação com o escalonador do SO (*baseline*). Em relação às diferentes estratégias de alocação de *thread*, a estratégia *random* fornece as melhorias mais significativas. É apresentado o melhor resultado entre as 20 configurações aleatórias que foram testadas nessa estratégia, conforme explicado anteriormente. A melhor configuração da estratégia *random* atingiu uma aceleração máxima de $2,55\times$ no SAVIME com 20000 *chunks* e de $1,87\times$ no SciDB com 25000 *chunks*. Esse resultado indica a necessidade de uma abordagem dinâmica, pois, entre algumas alocações de *threads* aleatórias, é possível encontrar o melhor desempenho em relação às outras estratégias estáticas. Por exemplo, a estratégia *shared* atinge somente 47% do desempenho máximo encontrado com a estratégia *random*.

A Figura 5.4 mostra os resultados do acesso às memórias remota e local durante a execução de um operador de *subarray*. O acesso à memória local tem menor latência em comparação com o acesso à memória remota. Isso significa que, quanto maior o valor de acesso à memória local, menores são os efeitos NUMA no SGBD multidimensional. A variação no acesso total à memória *DRAM* por meio do uso de estratégias de alocação de *threads* mostra que nem todas as estratégias alcançaram um bom uso das memórias *cache*. Isso evidencia a necessidade de pesquisar dados na *DRAM* mais vezes e com maior latência.

Para entender melhor a relação entre o *speedup* e os resultados dos acessos *DRAM*, observe os resultados para 100 *chunks* – gráficos mais à esquerda nas figuras. Pode-se ver uma relação clara entre o número total de acessos *DRAM* e o desempenho final. Isso ocorre devido à alta latência para obtenção de dados das memórias *DRAM* e indica um mau uso das memórias *cache*. Além disso, o aumento do número de acessos remotos agrega um segundo problema à redução do *speedup* final. A quantidade de acessos remotos também indica o quão bem o escalonador posicionou as *threads* nos núcleos NUMA.

Por outro lado, observando-se a estratégia *compact*, a alocação de *threads* apresentou variações no *speedup*, entregando o pior desempenho em relação ao *baseline* (que não posiciona as *threads*). Ao olhar para os acessos remotos e locais na Figura 5.4, nota-se que a estratégia *compact* reduziu o número de acessos remotos para SAVIME em 50%, exceto quando observamos 20.000 *chunks* em que o número de acessos remotos é significativamente mais alto. O motivo é que a operação de *subarray* executa *chunks* na maior parte do tempo e cada *thread* executa

chunks diferentes. Este fato aumenta a probabilidade de haver muitos *chunks* sendo processados por *threads* em simultâneo, o que também aumenta a contenção nas memórias *cache*.

Outro resultado relevante é que a *PetriNet* não apresentou tanto *speedup* quanto observado em um SGBD relacional. A *PetriNet* auxilia o SO na alocação de núcleos por meio da avaliação da carga da CPU. Ao contrário de um SGBD relacional, a carga da CPU no SGBD multidimensional mantém um alto uso da CPU [Lustosa et al. 2016]. Consequentemente, a *PetriNet* alocou todos os núcleos da arquitetura NUMA para o SGBD multidimensional SAVIME. No caso do SciDB, nota-se que ele não utiliza todos os núcleos de *CPU*. Como resultado, a alocação de núcleos realizada pela *PetriNet* não afeta muito o desempenho alcançado pelo *baseline* nesse caso.

5.3.1.1 Avaliando o impacto do número de chunks *alterando o* baseline

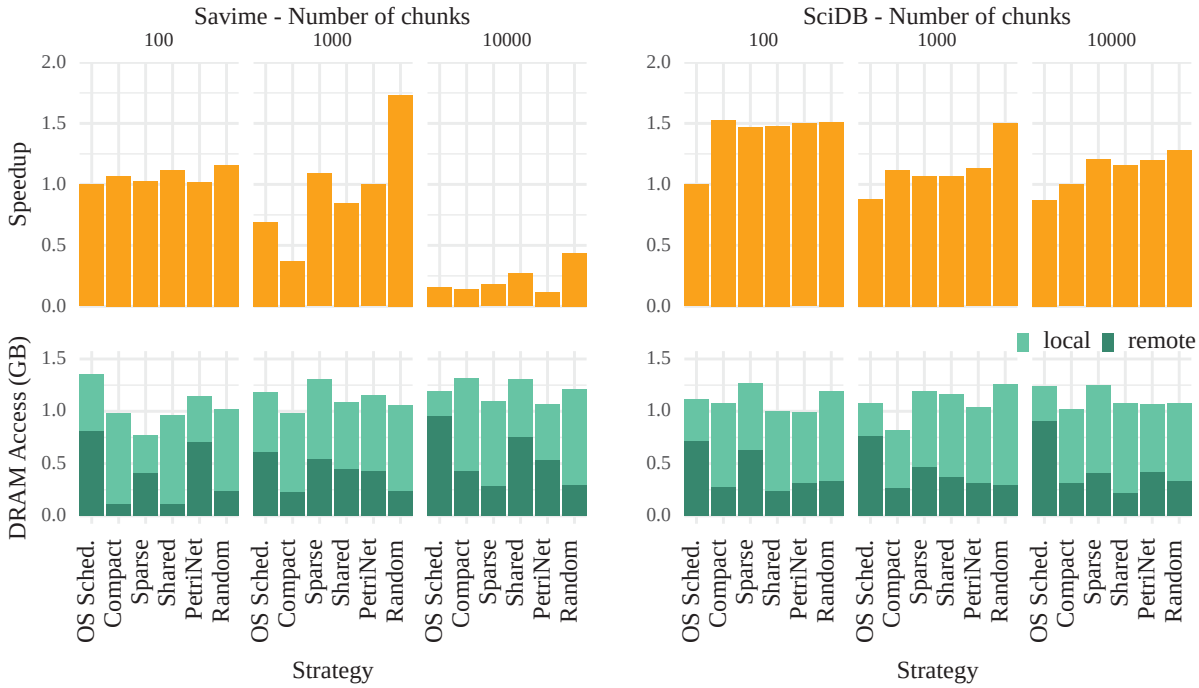


Figura 5.5: *Speedup* e a quantidade de acessos à memória comparando diferentes estratégias de alocação de *threads*, variando o número de *chunks*. Os experimentos executaram a operação *subarray* em um SGBD multidimensional de 1 GB, na máquina *Skylake*. O *baseline* utilizado foi o experimento de 100 *chunks*, com o agendamento do SO.

Para verificar os resultados por outra perspectiva, o *baseline* foi alterado para o experimento com 100 *chunks*, com a alocação de *threads* coordenada pelo SO. A Figura 5.5 mostra os resultados de *speedup* para o SAVIME e SciDB ao variar o número de *chunks*, por meio de diferentes estratégias de alocação de *threads*, em execução na máquina *NUMA-Skylake*. Os resultados são normalizados para o escalonador do SO (*baseline*) com 100 *chunks*. Para esse experimento específico, a estratégia *random* foi escolhida entre 20 mapeamentos aleatórios possíveis – a seção 5.3.3 apresenta a pesquisa exaustiva sobre todas as combinações.

A estratégia *random* produz o impacto mais positivo, atingindo um *speedup* máximo de até $1,7\times$ no SAVIME e $1,5\times$ no SciDB, ambos executando com 1000 *chunks*. Nesta análise, foi possível perceber que as estratégias de alocação de *threads* beneficiam-se da arquitetura *NUMA* quando usam um menor número de *chunks* (portanto, tamanhos de *chunk* maiores), possivelmente devido à quantidade reduzida de agendamento de *chunk* para as *threads* e a menor quantidade de

espalhamento de dados entre os nós *NUMA*. Esses resultados mostram que se pode alcançar, em média, 52% de ganhos de desempenho com novas abordagens de alocação de *threads*.

Também observa-se na Figura 5.5 que fixar cada *thread* em um núcleo específico é benéfico para o desempenho do sistema. Isso evita a migração de *thread* entre os nós NUMA que ocorre quando o SO tenta manter o equilíbrio da carga. Este resultado mostra a necessidade de uma abordagem dinâmica de alocação de *threads*.

A Figura 5.5 também apresenta a quantidade de acessos às memórias remota e local da operação do *subarray*. Os resultados indicam indiretamente o quão bem o escalonador posicionou as *threads*. Além disso, o acesso total à memória *DRAM* indica o quão bem as memórias *cache* foram utilizadas. A variação dos acessos *DRAM* mostra que as estratégias se beneficiaram, diferentemente da memória *cache*. A estratégia *random* mostra uma redução no acesso remoto de $4,9\times$ para o SAVIME e de $3,5\times$ para o SciDB. No entanto, não se pode encontrar um *link* direto entre os acessos *DRAM* e o *speedup* final, o que indica que mais métricas são necessárias para explicar completamente os resultados. A subseção 5.3.3 conduz uma avaliação mais ampla para entender melhor a influência dos acessos à *cache* e à memória no desempenho final.

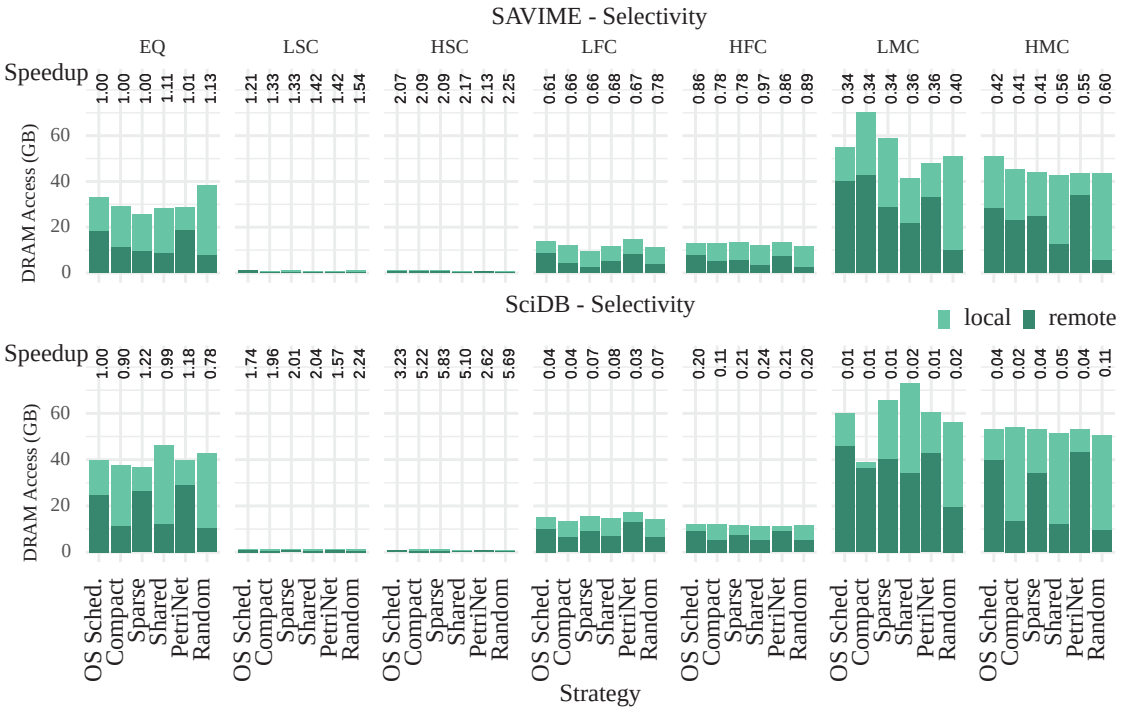


Figura 5.6: *Speedup* e quantidade de acessos à memória comparando diferentes estratégias de alocação de *threads*, variando a seletividade do operador. Os experimentos executaram a operação *subarray* em um banco de dados de matriz de 50 GB, na máquina *NUMA-Skylake*. O número do eixo *X* superior é o *speedup* de cada estratégia.

5.3.2 Impacto da seletividade

Nesta subseção, é avaliado o impacto de diferentes seletividades de consulta, usando um conjunto de dados de 50 GB. A seletividade indica a porcentagem de dados que precisa ser filtrada para materializar a saída do *subarray*. Alta seletividade (**H** — neste experimento, 70%) indica que a consulta filtra mais dados e, consequentemente, menos dados são materializados para a saída em *DRAM*. Baixa seletividade (**L** — neste experimento, 20%) indica o oposto. Também varia-se o número de *chunks*, conforme eles precisam ser transferidos através da hierarquia de

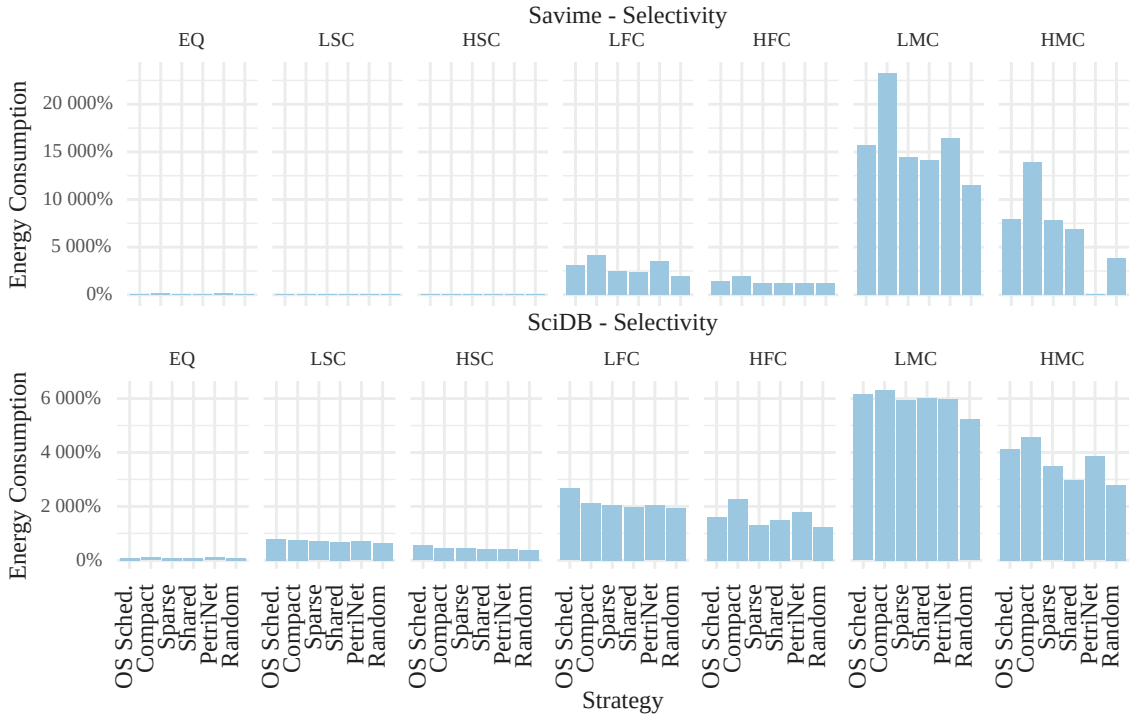


Figura 5.7: Consumo de energia na *DRAM* com banco de dados de 50 GB usando diferentes seletividades do operador nos SGBDs multidimensionais em *NUMA-Skylake*.

memória, para validar os filtros de consulta: *Single Chunk* (**SC**), *Few Chunks* (**FC**, aqui 20% dos *chunks*) e *Many Chunks* (**MC**, aqui 100% dos *chunks*). Além disso, é apresentada a consulta exata (**EQ**), que seleciona inteiramente um único *chunk*. O número total de 210 *chunks* foi utilizado para reproduzir a carga de trabalho executada no SAVIME [Lustosa et al. 2016] e armazenar o conjunto de dados de 50 GB em *chunks* tridimensionais ($250 \times 250 \times 500$) de atributos de valor *double*.

A Figura 5.6 apresenta o *speedup* e traz o número de acessos às memórias remota/local variando a seletividade. Primeiro, nota-se que ambos os SGBDs multidimensionais executam operações eficientes. Eles só acessam a memória relativa ao *chunk* solicitado usando suas coordenadas: observe a quantidade reduzida de acessos à memória para *LSC* e *HSC*. Comparando os três cenários com baixa seletividade – *LSC, LFC e LMC* – com seus respectivos pares com alta seletividade, a alta seletividade apresenta um desempenho melhor. Esse resultado é esperado, pois a alta seletividade filtra mais dados, reduzindo, assim, o uso da memória *cache* para armazenar os resultados e reduzindo ainda a pressão na memória *DRAM*.

Novamente, para esses resultados, é possível observar que nem o número remoto, nem o número local de acessos à *DRAM* parece se correlacionar diretamente com o *speedup* final. No entanto, observa-se que para quase todas as situações, existe um mapeamento diferente que apresenta melhor desempenho, com aumento de $2,25\times$ e $5,83\times$ no SAVIME e no SciDB, respectivamente. Além disso, esses mapeamentos com a estratégia *random* parecem ter uma tendência de apresentar baixos acessos remotos, com redução média de $5\times$ no SAVIME e de $4,1\times$ no SciDB.

A Figura 5.7 mostra os resultados do consumo de energia na memória *DRAM*, normalizado pelo escalonador do SO executando **EQ**. O comportamento apresentado é semelhante aos resultados do *speedup*. A estratégia aleatória reduziu, em média, a energia *DRAM* em 68% para o SAVIME e em 16% para o SciDB. Em ambos os bancos de dados, a estratégia de alocação

de *threads compact*, que mapeia as *threads* para núcleos em apenas um nó NUMA, aumentou o tráfego de memória devido à alta contenção de memória, que resultou em mais consumo de energia na memória principal.

No geral, o consumo de energia foi reduzido significativamente, por conta de diminuições no acesso remoto à memória e na movimentação de dados. A movimentação de dados é responsável pela maior parte do consumo de energia da memória.

5.3.3 Avaliação da estratégia *random*

Esta seção apresenta uma avaliação extensa das combinações de alocação de *threads*. Para isto, foi utilizada a máquina *NUMA-SandyBridge* por causa da sua reduzida quantidade de núcleos, o que permite avaliar todas as 462 combinações possíveis sem repetições de análogos. Devido à quantidade de núcleos de processamento para a máquina *NUMA-Skylake* (20 núcleos), um experimento exaustivo com 92378 combinações não seria possível nessa arquitetura. O SAVIME é executado com 12 *threads*, variando o número de *chunks* para uma matriz de 1 GB nestes experimentos.

Figura 5.8: Desempenho e acessos à *DRAM* para todas as combinações de alocação de *threads* na máquina *NUMA-SandyBridge*. O experimento usa SAVIME executando o operador *subarray* em um banco de dados de 1 GB. Os resultados são ordenados pelo aumento de velocidade em comparação com o escalonador do SO.

Ao definir o número total de combinações com a estratégia *random* de alocação de *threads*, consideram-se a hierarquia de memória da máquina e o número de *threads* usadas pelo SAVIME. Olhando para essas métricas, é utilizada a combinação simples sem repetição $\binom{n}{r} = \frac{n!}{r!(n-r)!}$. Com n *threads*, deve ser feita a escolha de r das mesmas para fazer a alocação em um nó da arquitetura NUMA que compartilha a memória *cache*. No *NUMA-SandyBridge*, encontra-se um total de 924 combinações executando 12 *threads*. No entanto, considerando que são dois nós NUMA idênticos, com 6 núcleos em cada nó, é reduzido para um total de 462 possibilidades de alocações de *threads* diferentes.

A Figura 5.8 mostra os resultados do *speedup* em todas as combinações de alocação de *threads* em três diferentes quantidades de *chunks*. Observa-se que os experimentos com 100 e 1000 *chunks* apresentam variações mais pronunciadas no *speedup*. É possível notar o claro impacto dos acessos remotos no mau desempenho (combinações entre 0 – 100, com pontos verdes maiores que azuis). Quando o mapeamento possibilita uma redução na quantidade de acessos remotos, pode-se observar um aumento no desempenho (combinações entre 100 – 200). Por fim, nota-se que o desempenho ainda se beneficia sempre que a memória *cache* é melhor utilizada, reduzindo assim a quantidade total de acessos à memória *DRAM* (combinações entre 200 – 462). Embora seja difícil distinguir benefícios de melhor uso de *cache* e menores acessos remotos, pode-se observar que ambas as métricas têm grande influência no desempenho final do SAVIME.

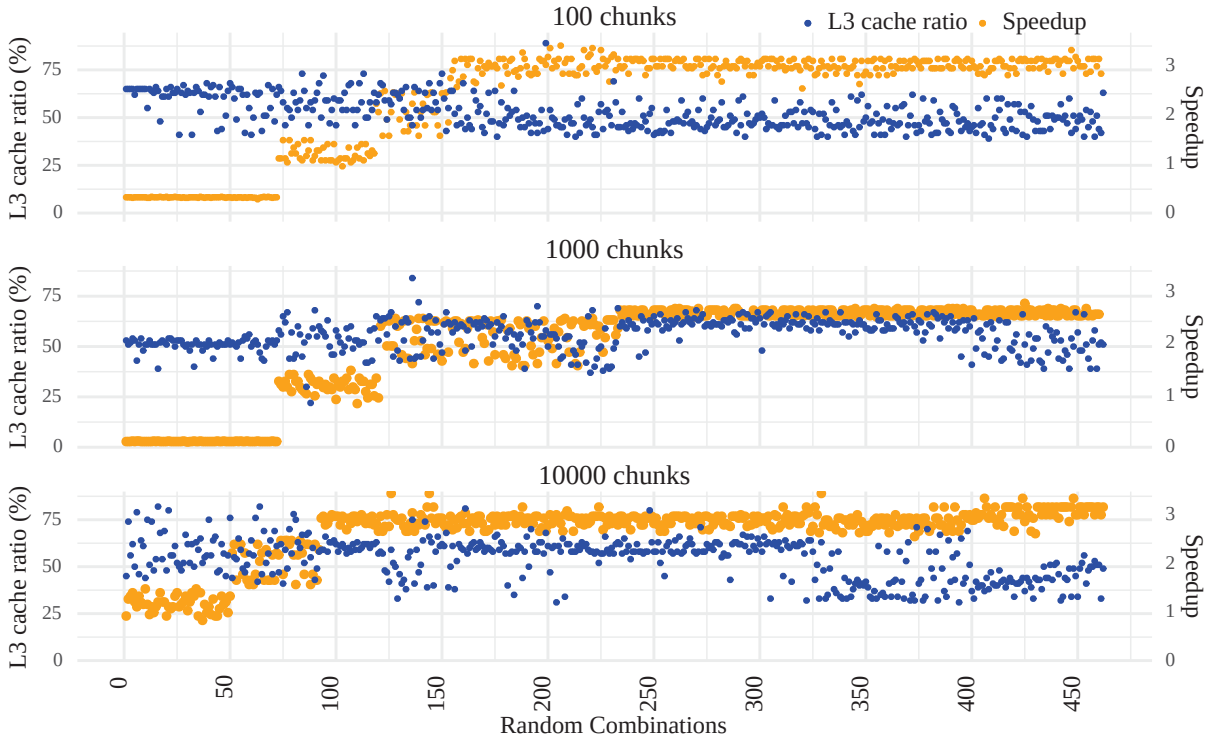


Figura 5.9: L3 *Cache miss ratio* em um banco de dados de 1 GB, com diferentes números de *chunks*, variando o número total de combinações de alocação de *threads* no *NUMA-SandyBridge*.

Nota-se que o *speedup* diminui em 7% do número total de experimentos realizados. Isso porque, em algumas combinações da estratégia *random*, as *threads* que trabalham no mesmo *chunk* acabam em nós diferentes, forçando acessos remotos com 2, 5× mais acessos à memória. A Figura 5.8 mostra o acesso às memórias local e remota.

A Figura 5.9 mostra que a *cache miss ratio* segue o mesmo padrão observado no acesso à memória. O desempenho aumenta à medida que a *cache miss ratio* do último nível é reduzida. Esses resultados corroboram os achados presentes nos últimos resultados plotados (Figura 5.8). Além disso, observa-se que o uso de menos *chunks* diminui o *cache miss*, possivelmente porque as células dos pequenos *chunks* cabem nos *caches*, evitando penalidades de *miss*.

5.3.4 Comparação de desempenho em arquiteturas NUMA

Este último experimento, com resultados apresentados na Figura 5.10, concentra-se na comparação de desempenho entre as duas máquinas NUMA (*NUMA-Skylake* e *NUMA-*

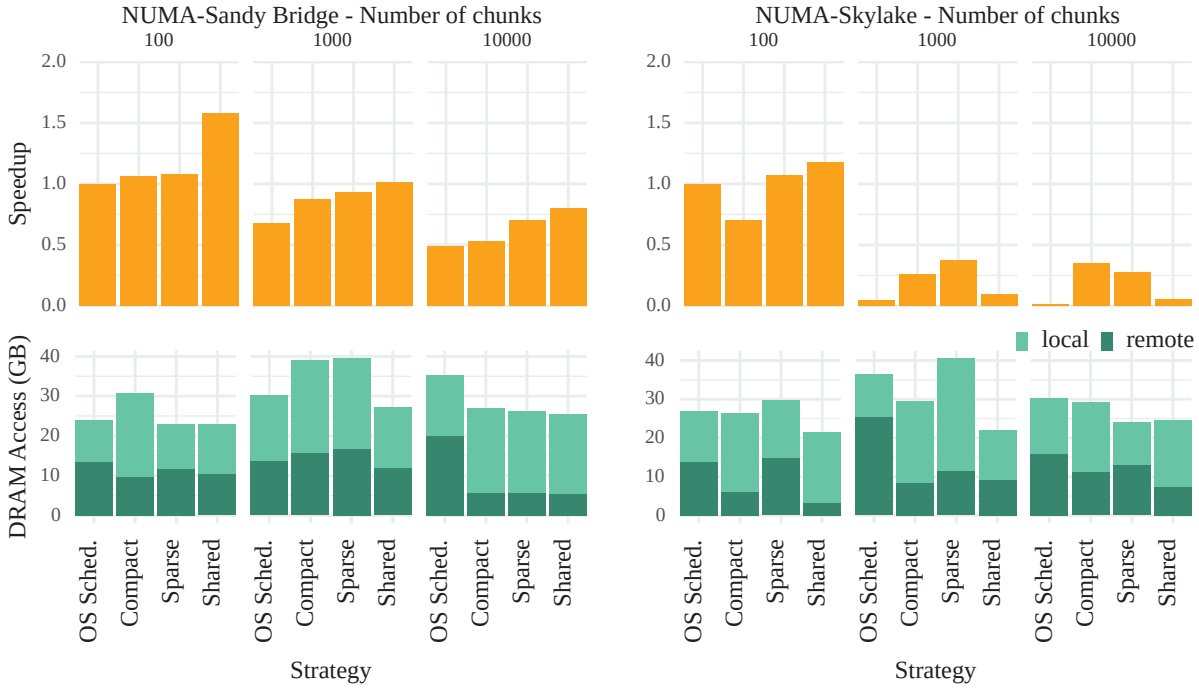


Figura 5.10: Comparação de desempenho da operação *subarray* em um banco de dados de 50 GB, usando diferentes números de *chunks*, no SAVIME SGBD multidimensional, nas arquiteturas *NUMA-Skylake* e *NUMA-SandyBridge*.

SandyBridge), executando o SAVIME. O objetivo é analisar a influência de diferentes estratégias de alocação de *threads*. Aqui, o número de *chunks* de uma matriz com 50 GB é variado. Os resultados são normalizados para os resultados do escalonador do SO com 100 *chunks* de cada máquina. As três principais diferenças entre as máquinas NUMA são o número de núcleos, o tamanho do *cache* L3 e a diferença entre as latências da memória local e remota (fator NUMA). A máquina *NUMA-Skylake* tem 2× 10 núcleos, compartilhando 14 MB de L3 e o fator NUMA de 1, 36. Já a *NUMA-SandyBridge* tem 2× 6 núcleos, compartilhando 15 MB de L3 e fator NUMA de 1, 32.

Primeiramente, é possível observar os diferentes impactos no *speedup* à medida que aumenta o número de *chunks*. A *NUMA-Skylake* mantém o *speedup* melhor do que o *NUMA-SandyBridge*. Além disso, o resultado mais interessante é que políticas de agendamento bem conhecidas fornecem resultados de desempenho diferentes dependendo da máquina, o que indica que as políticas de alocação devem evoluir com as arquiteturas mais novas, motivando também soluções sob medida para cada sistema NUMA.

5.3.5 Conclusões do efeito NUMA no SGBD de matriz

É possível concluir, a partir dos resultados apresentados, que as políticas de alocação de *threads* bem conhecidas fornecem *speedup* moderado e ainda estão longe do desempenho máximo possível para o SGBD multidimensional. Ao olhar para a seletividade, observa-se que ambos os sistemas de banco de dados são afetados por variações na seletividade, em que o desempenho é degradado começando da seletividade mais alta para a mais baixa.

Também é possível observar que encontrar uma combinação de alocação de *threads* que melhore o desempenho deve reduzir o acesso remoto e melhorar a utilização da hierarquia de memória *cache*. Os resultados mostram que uma quantidade semelhante de acessos à *DRAM*

pode levar a *speedups* muito diferentes (combinações 100 – 200 na Figura 5.8), o que pode dificultar o uso de heurísticas na busca do melhor mapeamento. No entanto, observa-se que 56% das combinações possíveis (combinações 200 – 462 na Figura 5.8) levam ao desempenho máximo – o que pode motivar os pesquisadores a sugerir melhores escalonadores para os SGBDs multidimensionais. Os resultados também motivam soluções customizadas para cada sistema NUMA, pois os sistemas podem responder diferentemente considerando seu fator NUMA e *cache* disponível.

5.4 CONSIDERAÇÕES FINAIS

Com base no fato de que os SGBD relacional e SGBD multidimensional adotam estratégias semelhantes ao usar o paralelismo *multi-thread*, apenas o primeiro teve seu comportamento de desempenho ao usar sistemas NUMA extensivamente estudado até então. Pelo que se sabe, este é o primeiro estudo sobre o impacto na aceleração e no consumo de energia do processamento de consultas de matriz em máquinas NUMA.

Ao implementar diferentes estratégias de alocação de *threads* em dois SGBDs multidimensionais, mostra-se como cada estratégia se comportou. Os resultados confirmam que a arquitetura NUMA afeta gravemente o desempenho da operação de *subarray*. A operação de *subarray* é baseada em condições de desigualdade e requer faixas móveis de células de matriz em nós de computação para validar essas condições. Mais uma vez observamos que a estratégia de alocação de *threads* de *baseline* com base no mapeamento de *thread* do SO não reconhece o relacionamento entre os operadores de consulta no plano de execução da consulta. Assim, as *threads* são posicionadas para equilíbrio de carga longe dos ganhos máximos possíveis. Também observa-se que diferentes arquiteturas NUMA implicam em ganhos de desempenho distintos.

No próximo capítulo, é apresentado um escalonador de *threads* para SGBDs multidimensionais com objetivo de melhorar o desempenho desses sistemas de banco de dados em máquinas NUMA de forma dinâmica e observando os padrões de acesso à memória das operações dos SGBDs multidimensionais.

6 NASARRAY: UM JOGO NÃO-COOPERATIVO PARA ALOCAÇÃO DE *THREADS* DE BANCO DE DADOS DE MATRIZ

As matrizes multidimensionais – que podem ser conhecidas como dados dados em grade ou cubos de dados – são primordiais em muitos domínios da ciência e da engenharia. Diariamente, a quantidade de dados que são produzidos a partir destes domínios é enorme. Na última década, a NASA acumulou cerca de 32 petabytes de dados científicos, já a ECMWF[1] possui 220 petabytes [Baumann et al. 2021]. Portanto, há uma necessidade de analisar e manipular da forma mais eficiente possível os dados organizados em matrizes multidimensionais. Os SGBDs multidimensionais são projetados para modelagem, armazenamento e processamento de matrizes multidimensionais. Espera-se que, em apenas alguns anos, a quantidade de dados científicos a ser processada e analisada aumente consideravelmente. Além disso, a necessidade do desenvolvimento de soluções eficientes irá crescer proporcionalmente ao volume de dados. Como descrito em *The Fourth Paradigm* [Hey et al. 2009] "A velocidade com que qualquer disciplina científica avança dependerá de como os pesquisadores colaboram entre si e com os tecnólogos em áreas da ciência como bancos de dados"[2](Hey et al. p. 287, tradução nossa).

Ao mesmo tempo, hoje, os sistemas multiprocessadores são peças fundamentais em *clusters* de computação científica. Idealmente, os SGBDs multidimensionais deveriam aumentar a velocidade linearmente ao usar sistemas multiprocessadores. No entanto, os sistemas multiprocessadores geralmente usam a arquitetura NUMA e os SGBDs multidimensionais não exploram adequadamente o seu potencial. Os SGBDs multidimensionais precisam mover grandes quantidades de dados durante as análises e enfrentam problemas como o aumento da latência no acesso remoto à memória. A penalidade de desempenho de acessos remotos à memória é significativa, principalmente quando a movimentação de dados é alta. Portanto, para evitar penalidades de desempenho e garantir o processamento de consultas com sucesso na arquitetura NUMA, é fundamental que haja uma distribuição cuidadosa de *threads* - conforme explicitado no capítulo anterior.

O SO usa estratégias de equilíbrio de carga para espalhar as *threads* por todos os núcleos, porém não assume características específicas da interação entre as operações em execução no banco de dados e a arquitetura do processador multi-núcleo. Neste capítulo, é detalhada a implementação de um mecanismo de escalonamento de *threads* para os SGBDs multidimensionais. A hipótese é que a movimentação de dados em nós NUMA pode ser mitigada mapeando as *threads* e fixando-as em núcleos de nós NUMA, com base no padrão de acesso à memória de cada operador de consulta. Assim, o escalonador de *threads* descrito neste capítulo analisa a operação em execução para determinar o padrão de acesso à memória atual e escolher o nó NUMA para o mapeamento e fixação da *thread*.

O mecanismo proposto analisa informações de falta de dados em *cache* coletadas por meio de contadores de *hardware* para definir as estratégias de alocação de *threads* entre nós NUMA. Baseado na teoria dos jogos, o mecanismo mostra que é possível definir um esquema de alocação de *threads* que usa os padrões de acesso à memória e converge para resultados ótimos. Para avaliá-lo, são analisados o *speedup*, os acessos às memórias local e remota e o consumo de energia, a partir da métrica *Energy-delay product* (EDP).

Em resumo, as contribuições deste capítulo são as seguintes:

---

[1]European Centre for Medium-Range Weather Forecasts

[2]"*The speed at which any given scientific discipline advances will depend on how researchers collaborate with one another, and with technologists, in areas of science such as databases*"

- **Mecanismo de alocação de *threads*:** é proposto um mecanismo para alocação de *threads* em máquinas NUMA com base nos padrões de acesso à memória e nos falta de dados em *cache* por SGBDs multidimensionais. O mecanismo coleta informações do SO e do SGBD multidimensional para tomar a decisão da alocação de *threads*.
- **Análise de diferentes operações de SGBDs multidimensionais:** realiza-se uma análise experimental do impacto da arquitetura NUMA em diferentes operações de consulta.
- **Modelo de custo de alocação**: é descrito um modelo de custo de alocação de *thread* em um nó NUMA baseado em falta de dados em *cache*.
- **Melhorias de desempenho**: verifica-se, experimentalmente, que o mecanismo melhora o tempo total de execução e reduz os acessos à memória remota na arquitetura NUMA em dois diferentes SGBDs multidimensionais.

Este capítulo está organizado da seguinte forma: a próxima seção descreve os SGBDs multidimensionais e suas operações de consultas, assim como a arquitetura NUMA e o impacto no desempenho dos SGBDs multidimensionais. A seção 6.2 descreve o modelo matemático da teoria dos jogos utilizado no mecanismo e apresenta uma visão geral do mecanismo de alocação de *threads*. A seção 6.3 mostra os resultados empíricos e a seção 6.4 traz as considerações finais.

6.1 BANCO DE DADOS DE MATRIZ MULTIDIMENSIONAL

Esta seção descreve os operadores de consulta do SGBD multidimensional utilizados para mapear os padrões de acesso à memória. Além disso, apresenta uma discussão sobre os problemas de desempenho gerados pela movimentação de dados durante a execução de *threads* dos SGBDs multidimensionais em uma máquina NUMA.

6.1.1 Banco de dados de Matriz Multidimensional

O SGBD multidimensional implementa o modelo de dados de matriz n-dimensional. As várias dimensões simplificam o acesso e a análise de dados por meio de diferentes visualizações. Nesse modelo, cada célula pertencente à matriz contém atributos com o mesmo tipo de dados. Os SGBDs multidimensionais usam linguagens de consultas funcionais estendidas para operações multidimensionais especiais, como operações geométricas, lineares e de agregação. Uma matriz é dividida em muitas partes nomeadas de *chunks*.

Neste capítulo, são utilizados dois SGBDs multidimensionais: o SAVIME e o SciDB. Ambos trabalham com o processamento de *chunks* para chegar ao resultado de uma consulta, bem como recorrem ao paralelismo para melhorar o desempenho da consulta, usando os recursos disponíveis no *hardware*. No SciDB, observa-se o paralelismo intra-consulta e inter-consulta. Já no SAVIME o paralelismo presente é o intra-consulta, sendo que cada operador de consulta é processado por várias *threads* e não é possível o processamento de duas consultas em simultâneo.

Cada um dos SGBDs multidimensionais permite consultas que são realizadas, com base nas propriedades e conteúdos da matriz, usando linguagens declarativas. As linguagens declarativas garantem um alto grau de flexibilidade na formulação das consultas e na otimização interna de consultas. O processamento de matrizes é a funcionalidade central tanto no SAVIME quanto no SciDB, sendo possível realizar desde um simples redimensionamento até estatísticas e álgebra linear geral.

O modo de armazenamento e processamento de consulta não difere em ambos os SGBDs multidimensionais, SAVIME e SciDB. Semelhantemente, os *chunks* são processados por meio de chamadas de funções aninhadas e que geram um fluxo de *chunks* processados em *pipeline*. Em adicional, o SAVIME combina o modelo de *pipeline* com a materialização para tentar suprimir localidade de dados insatisfatórias causadas pelo modelo de *pipeline* e problemas com a materialização de grandes quantidades de dados. Durante uma consulta, a organização da matriz facilita a busca pelos dados. A ordem implícita das células de uma matriz densa oferece uma busca mais rápida para algumas operações por meio da disposição linear dos dados.

Durante o processamento de consultas, o SAVIME e o SciDB utilizam a mesma premissa para o paralelismo. No SAVIME, são processados grupos de *chunks*, cada um com uma quantidade pré-estabelecida de *threads*. No SciDB, um número de *threads* é definido para processar os *chunks* por instâncias. O modo de operação é semelhante, o que difere é que a divisão utilizada pelo SciDB é realizada para facilitar o processamento dos *chunks* em nós de computação diferentes, sendo essas arquiteturas *shared-nothing* em que não são compartilhadas entre os nós de computação a memória ou armazenamento.

O SAVIME e o SciDB possuem uma quantidade substancial de operadores de consulta. Um deles, discutido no capítulo anterior, é o *subarray*. Na subseção seguinte, serão descritos os principais operadores de consulta utilizados para validação do mecanismo proposto.

6.1.2 Padrões de acesso à memória no banco de dados de matriz multidimensional

Os operadores de consulta no SAVIME e no SciDB executam operações em matrizes multidimensionais. Algumas dessas operações trabalham utilizando os índices das matrizes e projetam resultados que são baseados em intervalos pré-definidos. As principais operações presentes no SAVIME e no SciDB estão destacadas na Tabela 6.1. É possível verificar as diferentes operações e, baseado na operação, analisar os diferentes padrões de acessos à memória.

Para definir os padrões de acesso à memória, foram utilizadas as características apresentadas em [Kepe 2019]. O autor classifica o acesso à memória baseado no padrão apresentado pela operação e apresenta uma taxonomia de operadores de consulta. Outros autores discutiram os impactos dos acessos à memória em arquiteturas modernas [Manegold et al. 2002; Zeuch e Freytag 2015; Müller e Plattner 2012; Ozisikyilmaz et al. 2006]. Nesse contexto, para designar o mecanismo apresentado nesta tese, são utilizadas essas premissas para caracterizar as operações dos SGBDs multidimensionais a partir da análise dos padrões de acesso à memória.

Os acessos à memória são classificados por [Kepe 2019] em quatro grupos, em que são consideradas as seguintes características: operações com alto reuso de dados e com baixo reuso de dados; operações que possuem acesso à memória não coalescente e aquelas que possuem acesso à memória coalescente (i.e. contíguo). Assim, levando em conta esses critérios, foram analisadas três operações diferentes neste trabalho. A primeira operação é a discutida no capítulo 5, o *subarray/subset*, uma operação que cria uma fatia de dados n-dimensional a partir de uma especificação de intervalos. A operação de *subarray* pode ser classificada como baixo reuso de dados com acesso à memória coalescente (**BC**) o que acontece em $2/3$ das dimensões. Os dados são projetados e podem ser aproveitados dados que estão em *chunks* próximos. Entretanto, os dados já projetados não são mais utilizados durante a operação, caracterizando baixo reuso de dados.

Outra operação observada nesta tese é a agregação. Essa operação, em ambos os SGBDs multidimensionais, pode ser aplicada em diferentes dimensões para produzir os resultados. Na Figura 6.1, é possível observar a agregação sendo aplicada por meio de uma soma. Ela gera uma nova matriz, resumindo valores das colunas e das dimensões. No SAVIME, os *chunks*

Tabela 6.1: Operadores de Consulta em SGBDs multidimensionais.

Operador	**Descrição**	**Savime**	**SciDB**
ADDDIM	Adiciona uma dimensão na matriz	✓	✓
AGGREGATE	Resume os dados avaliando funções de agregação comuns.	✓	✓
BERNOULLI	O operador bernoulli avalia cada célula, gerando um número aleatório e vendo se ele está no intervalo (0, probabilidade).		✓
BETWEEN	Seleciona os dados da matriz da região especificada.		✓
CROSS_JOIN	Cria o produto cartesiano entre células em duas matrizes.	✓	✓
DERIVE	Adiciona um atributo com valores derivados calculado a partir de uma expressão definida pelo usuário.	✓	✓
DIM_JOIN	Equivalente a um equijoin considerando pares de índices de dimensão correspondentes.	✓	✓
INVERSE	O operador inverso produz o inverso da matriz de uma matriz quadrada.		✓
PROJECT	Projeta a matriz de entrada nos atributos especificados, na ordem especificada.		✓
SCAN/SELECT	O operador scan exibe o conteúdo de cada célula em uma matriz.	✓	✓
SLICE	O operador slice obtém uma amostra de células ao longo de um plano especificado de uma matriz.		✓
SORT	Classifica uma matriz por um ou mais atributos.		✓
SUBARRAY/SUBSET	Cria uma fatia de dados n-dimensional, de acordo com os limites especificados.	✓	✓
TRANSPOSE	Inverte a ordem das dimensões		✓
WHERE/FILTER	Filtra dados de acordo com um predicado lógico complexo, considerando atributos e atributos.	✓	✓

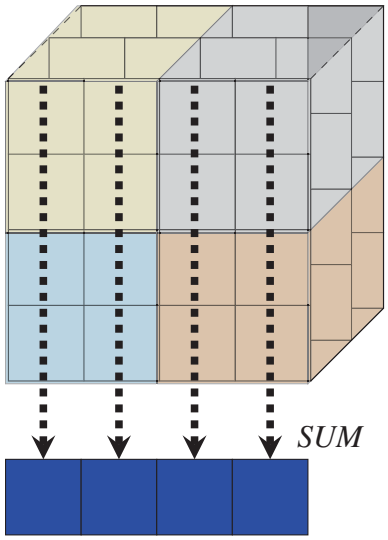


Figura 6.1: Exemplificação da operação de agregação em uma matriz multidimensional em um SGBDs multidimensionais.

são varridos sequencialmente e os resultados são salvos em uma memória temporária (*buffer*). Após o processamento de todos os *chunks*, os resultados que estão nas memórias temporárias são unidos em apenas um resultado [Lustosa 2020]. No SciDB, a agregação vai diretamente na célula que é avaliada independentemente da posição do *chunk*. A operação de agregação pode ser classificado como alto reuso de dados (**AR**). Além disso, difere do *subarray* porque apresenta acesso à memória não coalescente.

Por fim, será considerada neste trabalho a operação de junção (*join*). Nos SGBDs multidimensionais, a junção é realizada a partir das dimensões. Na Figura 6.2, é possível observar a junção de duas matrizes. O resultado de uma operação de junção é uma nova matriz com duas colunas. A matriz resultante também pode ter operadores de matriz adicionais aplicados a ela. A junção no SAVIME é realizada por meio de um laço aninhado em pares de *chunks* para combinar os dados de *chunks* que se cruzam. Posteriormente, a operação realiza uma junção de ordenação para combinar os *chunks*. O operador de junção no SciDB precisa unir todas as dimensões de um *chunk*. As identificações das dimensões e os valores são vistos como colunas, que produzem uma nova matriz como saída. Por ler e processar diferentes identificações de células e utilizar laço aninhado, a operação de junção é reconhecida como uma operação com alto reuso de dados na matriz menor e com acesso à memória coalescente (**AC**).

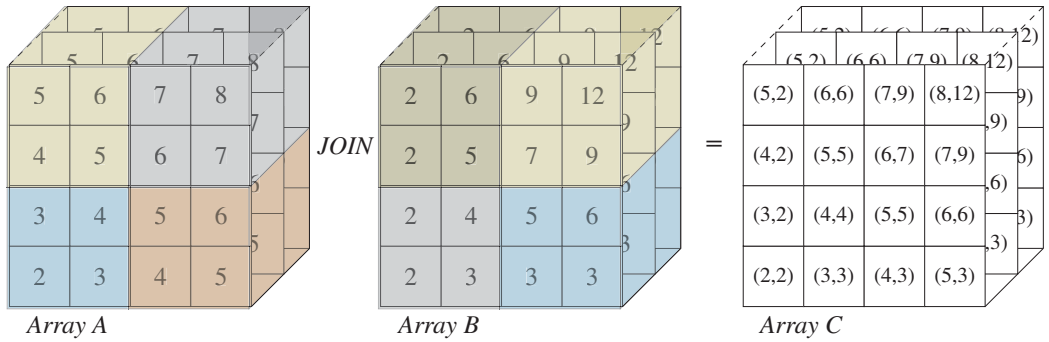


Figura 6.2: Exemplificação da operação de junção em um SGBDs multidimensionais. A junção une os valores da matriz A e B. Como resultado, tem-se a matriz C.

6.1.3 Análise dos efeitos da arquitetura NUMA no SGBD multidimensional

Nesta subseção, é avaliado o efeito da movimentação de dados na arquitetura NUMA, mostrando estatísticas de desempenho e uso de recursos dos SGBDs multidimensionais. Foi utilizado um *microbenchmark*, com uma base de dados de 1 GB, alterando a quantidade de *chunks*. A carga de trabalho possui uma matriz densa para as operações de agregação e *subarray*, e duas matrizes densas para a operação de junção. A carga de trabalho é baseada em dados do *benchmark* sísmico *HPC4e BSC* [Center 2016], usado no trabalho apresentado em [Lustosa e Porto 2019]. São utilizados os SGBDs SAVIME e SciDB. O número de *threads* foi escolhido baseado na quantidade de núcleos físicos disponíveis. O SciDB opera com 4 instâncias e com 5 *threads* para cada instância, totalizando 20 *threads*. Já o SAVIME opera em 4 *chunks* em paralelo, usando 5 *threads*. Para medir o desempenho do *hardware*, é utilizado o PCM [Willhalm Thomas 2012]. Neste experimento, foi executado o *microbenchmark* na máquina *NUMA-Skylake*), que possui dois nós, cada nó com um *Intel Xeon Silver* 4114 (com microarquitetura *Skylake*). A máquina inclui 128 GB de memória principal DDR-4.

6.1.3.1 Impacto do acesso remoto a memória

O primeiro experimento mostra que o escalonador do SO atual não é ideal para sistemas NUMA que executam consultas de SGBDs multidimensionais, levando a um grande número de acessos à memória remota, nas operações de *subarray*, agregação e junção. São comparados o escalonamento de *threads* apresentado no capítulo 5 (*Compact*, *Sparse*, *Shared*) com aquele executado pela política do SO não-modificado (*OS Sched.*). A operação de *subarray* passa por todos os *chunks* projetando metade deles. A junção opera em todos os *chunks* de duas matrizes

e em todas as dimensões. A agregação aplica uma soma somente na última dimensão. Neste experimento, é variado o número de *chunks* para valores menores, próximos à quantidade de núcleos disponíveis na arquitetura. Como já observado nos SGBDs relacionais a hipótese desta tese é mantida que o número de núcleos influencia na quantidade de *chunks*. Sendo assim, os resultados são normalizados pelo experimento com 20 *chunks* e escalonamento realizado pelo SO.

As Figuras 6.3, 6.4 e 6.5 comparam o escalonador do SO com as estratégias de alocação de *threads*. Observa-se que, em todas as operações analisadas de *subarray*, agregação (*aggregate*) e junção (*join*), o escalonador do SO apresenta mais acessos à memória comparado aos modos de alocação. Tal comportamento era esperado porque o escalonador do SO tenta manter o equilíbrio de carga e, a cada novo *chunk* processado em paralelo, foi observado que o SO tenta redistribuir a carga de trabalho ao longo dos nós.

Analisando as operações em execução, a operação de *subarray* seleciona metade da *matriz* e processa todos os *chunks*. Essa operação apresenta uma quantidade menor de acessos à memória devido à projeção direta dos valores, sem necessidade de carregar todos os dados não-relevantes à consulta para a memória. No SciDB, os acessos à memória são maiores. Para fazer a operação de *subarray*, o SciDB precisa reorganizar os dados e criar um *chunk* com os resultados, não somente projetá-los, – o que aumenta os acessos à memória pela necessidade de reunir todos os dados processados.

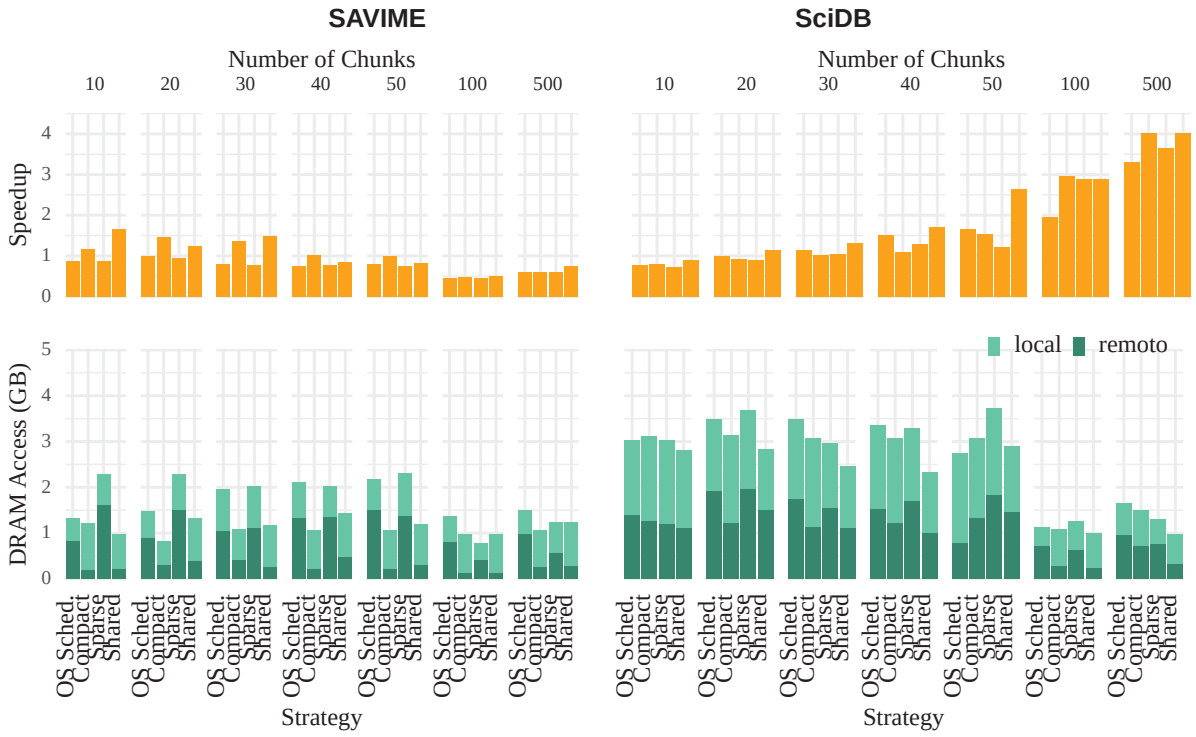


Figura 6.3: Impacto da arquitetura NUMA na execução da operação de *subarray*, no SAVIME e no SciDB, variando o número de *chunks*.

Já a operação de agregação no SAVIME aloca todos os dados em memórias temporárias para posterior leitura. A agregação trabalha apenas na dimensão especificada, não operando nas demais. O SciDB acessa os dados por meio das células e resolvendo a agregação. Além disso, o SciDB é ideal para executar em *clusters*, pois a execução de todas as instâncias na mesma máquina faz com que os acessos aos dados ocorra em todas as instâncias – que são quatro no caso dos experimentos desta tese. Os dados de todas as instâncias são lidos e reunidos para projetar os

resultados. A migração de *threads* do *OS* afeta negativamente o desempenho, devido ao fato de haver alocação de *threads* distantes dos dados.

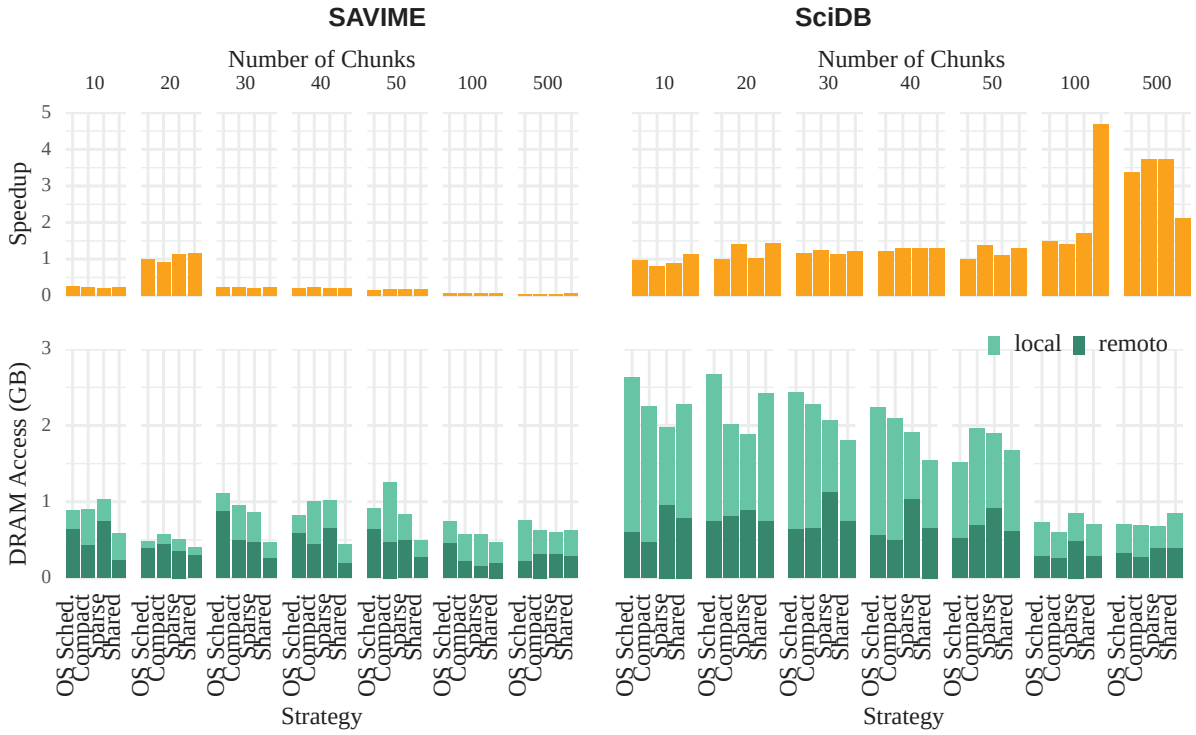


Figura 6.4: Impacto da arquitetura NUMA na execução da operação de agregação no SAVIME e SciDB, variando o número de *chunks*.

A operação de junção apresenta mais acessos à memória no SAVIME. Essa operação tem mais trabalho a ser executado, visto que carrega todas as informações para poder reorganizá-las. A junção tem um perfil alto de reuso de dados e o escalonamento de *threads* do SO força um número maior de acessos à memória remota. Isso ocorre porque, durante o processamento dos *chunks*, as *threads* precisam processar todo conteúdo dos *chunks*, mesmo usando apenas os índices das células. No SciDB, essa operação segue um comportamento semelhante, mesmo sendo menos flexível que o SAVIME quando se trata de junção. Isso não muda o fato que a movimentação de dados causada pela migração afeta o desempenho.

Por fim, outro fato interessante no SAVIME é que os melhores resultados são quando o número de *chunks* é igual à quantidade de núcleos físicos na *NUMA-Skylake*. Quantidades maiores de *chunks* fazem as *threads* mudarem constantemente os dados que estão processando, aumentando a movimentação de dados. Já o SciDB apresenta comportamento diferente, com os melhores resultados quando existem mais *chunks*. Supostamente, a divisão dos *chunks* em instâncias tem um desempenho melhor quando o tamanho do *chunk* fica menor. Entretanto, o movimento de *threads* entre os nós NUMA pelo *SO* causa um número maior de acessos remotos e acessos à memória principal, indicando pouco aproveitamento das memórias *cache*.

6.1.3.2 Impacto do escalonamento de threads *do SO*

Nesta subseção, é investigado o escalonamento de *threads* realizado pelo SO em cada padrão de acesso à memória. A Figura 6.6 mostra a migração de *threads* geradas pelo SAVIME para executar os diferentes padrões de acesso à memória, em uma base de dados de 1 GB com 100 *chunks*. As cores diferentes indicam os nós NUMA diferentes e os tons indicam os núcleos

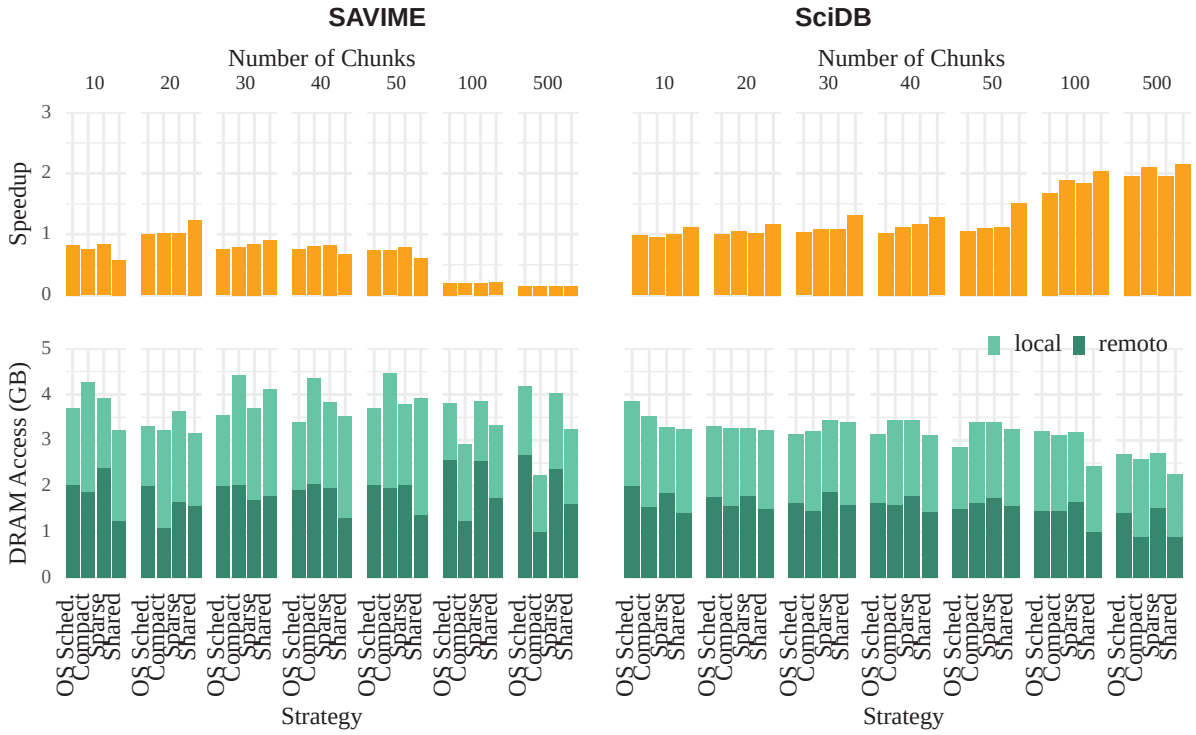


Figura 6.5: Impacto da arquitetura NUMA na execução da operação de junção, no SAVIME e SciDB, variando o número de *chunks*.

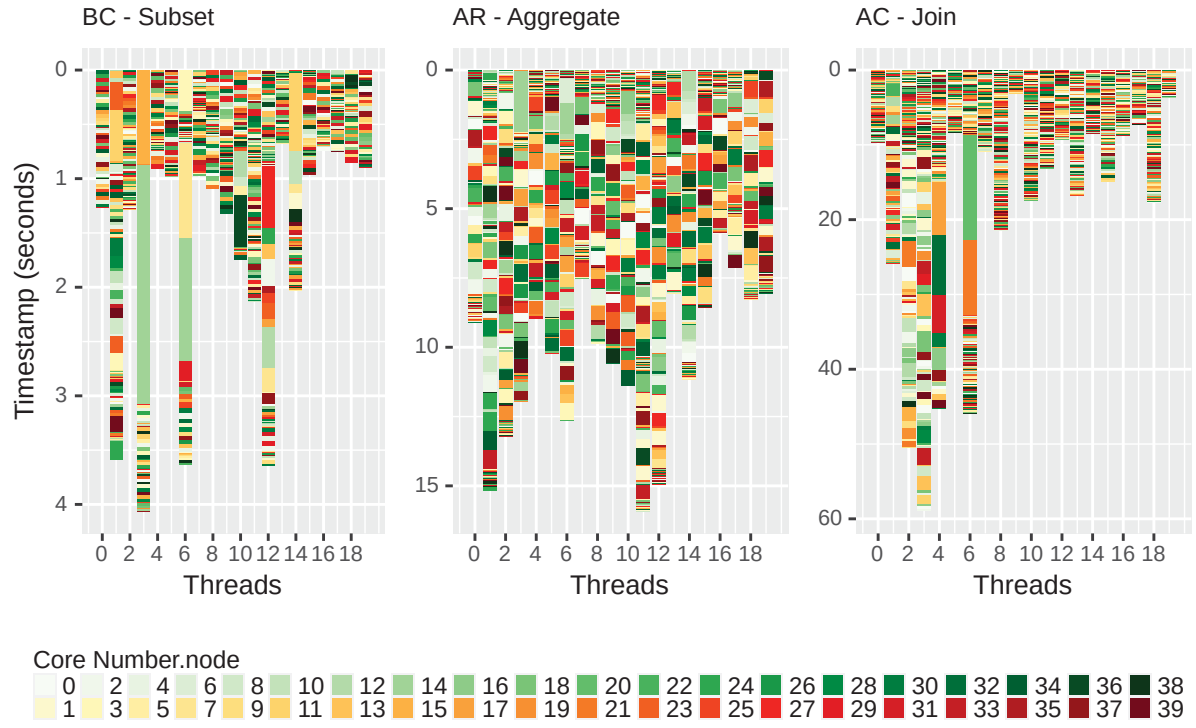


Figura 6.6: Avaliando o *lifespan* e a migração entre os núcleos das *threads* gerados pelos padrões de acesso à memória, em uma execução de cliente único, com todos os 20 núcleos disponíveis no SGBD multidimensional SAVIME.

de processamento no mesmo nó. É possível observar que as *threads* migram diversas vezes ao

longo da execução das operações. Isso indica que o SO tenta manter o equilíbrio de carga entre os nós NUMA e, consequentemente, gera custos adicionais de latência.

Outro ponto importante é que, nos resultados das três operações, existem algumas *threads* que apresentam um tempo maior. Essas são as *threads* definidas no primeiro nível de paralelismo. O SAVIME opera os *chunks* baseado em dois níveis de paralelismo. Por exemplo, no caso do experimento, é possível operar quatro matrizes paralelas e criar cinco *threads* para cada *chunk*, sendo o produto dessa relação o total de *threads* que o SAVIME pode utilizar. Observa-se que as *threads* do primeiro nível designam os *chunks* para serem processados e trabalham para completar outras operações necessárias para retornar o resultado.

A definição de níveis de paralelismo cria conjuntos de *threads* que trabalham sob um mesmo *chunk* e o esforço do SO em manter o balanceamento de carga pode afetar diretamente esses conjuntos. As *threads* são movidas ao longo da arquitetura NUMA e, como consequência, são afetadas pelos custos adicionais de latência de acessos remotos aos dados. Além disso, a agregação apresentou muitos acessos à memória principal. Isso mostra que a movimentação das *threads* ao longo dos nós NUMA gera muitos acessos remotos e faltas de dados na *cache* para gerenciar as memórias temporárias que armazenam os resultados de cada *chunk*.

6.2 NASARRAY: UM JOGO NÃO-COOPERATIVO PARA ALOCAÇÃO DE *THREADS* DE BANCO DE DADOS DE MATRIZ

A teoria dos jogos é uma teoria matemática que facilita a tomada de decisões. A proposta que apresentamos nesta tese é um algoritmo dinâmico e *online* baseado na teoria dos jogos motivado por considerar cada *thread* do SGBD multidimensional como tomador de decisão. Cada nó NUMA apresenta uma vantagem que varia com o tempo, conforme o consumo de recursos de *hardware*. Uma coleção de tomadores de decisão ocupa o nó NUMA. Cada tomador de decisão – chamado de agente – tem uma sequência associada de decisões independentes. No início da decisão de um agente, ele faz análises para verificar o seu lucro, que depende do número de outros agentes em sua localização atual e o nível de custo em cada nó NUMA. A ideia básica é modelar a alocação de *threads* na arquitetura NUMA como um equilíbrio de Nash em um jogo não-cooperativo [Nash 1951]. Nesta seção, é descrito o equilíbrio de *Nash* e apresenta-se uma visão geral do algoritmo.

6.2.1 Equilíbrio de Nash

O equilíbrio de Nash foi introduzido em 1950 por John Forbes Nash [Nash 1951]. O principal objetivo de Nash era encontrar o equilíbrio em um jogo não-cooperativo, analisando uma solução para diversos tomadores de decisão. Em um equilíbrio de *Nash*, os jogadores – tomadores de decisão – tomam decisões independentes que convergem para o melhor resultado global que é o Equilíbrio de Nash. As decisões são chamadas de estratégias e a estratégia de um jogador influencia outros jogadores. Cada jogador escolhe uma estratégia para atingir um resultado com o menor custo possível. Quando nenhum jogador tem incentivo para mudar de estratégia, o jogo encontra o equilíbrio de *Nash*. Formalmente, é definido como:

Seja (S, f) um jogo com n jogadores, onde:

- $S = S_1 \times S_2 \times \ldots \times S_n$ é o conjunto de estratégias de um perfil;
- Jogador $i \in 1, \ldots, n$;
- $f(x) = f_1(x), \ldots, f_n(x)$ é o conjunto de perfis de custo;

- Uma função de custo é avaliada em $x \in S$;
- x_i é um perfil de estratégia do jogador i e x_{-i} é um perfil de estratégia de todos os jogadores, exceto o jogador i;
- Cada jogador $i \in 1, ..., u$ escolhe uma estratégia x_i, resultando em um perfil de estratégia $x = (x_i, ..., x_u)$ então o jogador i com custo $f_i(x)$;
- Um perfil de estratégia $x^* \in S$ é um equilíbrio de Nash $\forall i, x_i \in S_i$, isto é, $f_i(x_i^*, x_{-i}^*)) \geq f_i(x_i, x_{-i}^*)$;

Quando o conjunto de estratégias está em equilíbrio de *Nash*, possivelmente encontram-se as melhores soluções para o jogo. O jogo pode ter muitos equilíbrios de Nash. No mecanismo, o jogo envolve as estratégias de posicionamento das *threads* nos nós NUMA.

6.2.2 Visão geral do mecanismo

Esta seção descreve o mecanismo dinâmico proposto para alocação de *threads* de processamento de consultas dos SGBDs multidimensionais, usando uma estratégia dinâmica baseada em Equilíbrio de *Nash*. Como visto anteriormente, a arquitetura NUMA afeta o desempenho de um SGBDs multidimensionais. O gerenciamento das *threads* pelo SO movimenta as *threads* por todos os nós NUMA, acrescentando a todos os acessos custos adicionais de latência.

O processamento de consulta paralela em SGBDs multidimensionais usa múltiplas *threads*. Naturalmente, as *threads* podem assumir o papel de tomadores de decisão. Na arquitetura NUMA, a *thread* tem múltiplos núcleos que estão disponíveis para o processamento das consultas ao longo dos nós. A alocação da *thread* pelo escalonador do SO na arquitetura NUMA pode impactar negativamente no desempenho do processamento de consultas. Nesse sentido, as *threads* tornam-se tomadores de decisão que, com base no tipo de acesso à memória da operação em processamento, podem escolher uma estratégia de alocação nos nós/núcleos NUMA.

Do ponto de vista da teoria dos jogos, a alocação de *threads* em uma arquitetura NUMA pode ser considerada um jogo. As *threads* são os jogadores e os padrões de acesso à memória definem todas as estratégias para determinar onde cada *thread* é posicionada. Portanto, várias *threads* escolhem o melhor nó para se posicionar. No contexto do Equilíbrio de *Nash*, os SGBDs multidimensionais possuem n *threads* $T = 1, 2, 3, ..., n$. Uma *thread* i é vinculada a um determinado nó NUMA, sendo o nó γ_{node} e a vinculação $\epsilon_i \in \gamma_{node}$. Seja ainda $\epsilon = \{\epsilon_i\}$ o perfil de alocação em cada nó e δ o conjunto de todos os perfis.

Formalmente, o equilíbrio de *Nash* em um jogo de alocação de *threads* pode mudar ao longo da carga de trabalho em execução. A situação de equilíbrio de *Nash* denota um possível alocação que satisfaz várias *threads*. Ou seja, as estratégias adotadas por todas as *threads* são consideradas a alocação ideal em um determinado momento do jogo. Para caraterizar o jogo, é necessário definir o objetivo a ser considerado pelas *threads*. Neste caso, seria minimizar a quantidade de *cache miss* de todas as *threads*, ou seja:

$$\min_{\epsilon \in \delta} f(\epsilon) = \sum_{i=1}^{n} m_i(\epsilon)/n \tag{6.1}$$

onde, m_i representa as faltas de *cache* da *thread* i sob a alocação $\epsilon \in \delta$, sendo que m_i dependerá do perfil de atribuição ϵ. No entanto, os detalhes de $m_i(\epsilon)$ são medidos utilizando contadores de *hardware*. As medições de faltas de *cache* são realizadas apenas das *threads* que já estão no

jogo. Com esses valores, o jogo de alocação de *threads* reage às medições de *cache* e decide a alocação. Como o mecanismo se trata de um jogo estratégico baseado em teoria de *Nash*, o jogo é representado pela tupla $< T, \delta, m_i i >$. Esse jogo tem enfoque em equilíbrios de *Nash* puros, ou seja, o perfil δ^* em que nenhuma *thread* teria o incentivo para mudar para um núcleo de um nó *NUMA* diferente. Essa alocação δ^* é um equilíbrio de *Nash* se $m_i(\epsilon_i^*, \epsilon_{-i}^*)) \geq m_i(\epsilon_i, \epsilon_{-i}^*)$ $\forall i, \epsilon_i \in \epsilon$. Em outras palavras, quando todas as *threads* estiverem na melhor alocação, trata-se de um equilíbrio de *Nash* – assumindo, é claro, que quando uma nova *thread* é iniciada, o equilíbrio de *Nash* pode se modificar. A importância disso reside no fato de que existe um conjunto de equilíbrios de *Nash* em que todas as *threads* têm um bom desempenho, ou seja, seu custo – o total de *cache miss* – é menor do que o custo do nó NUMA.

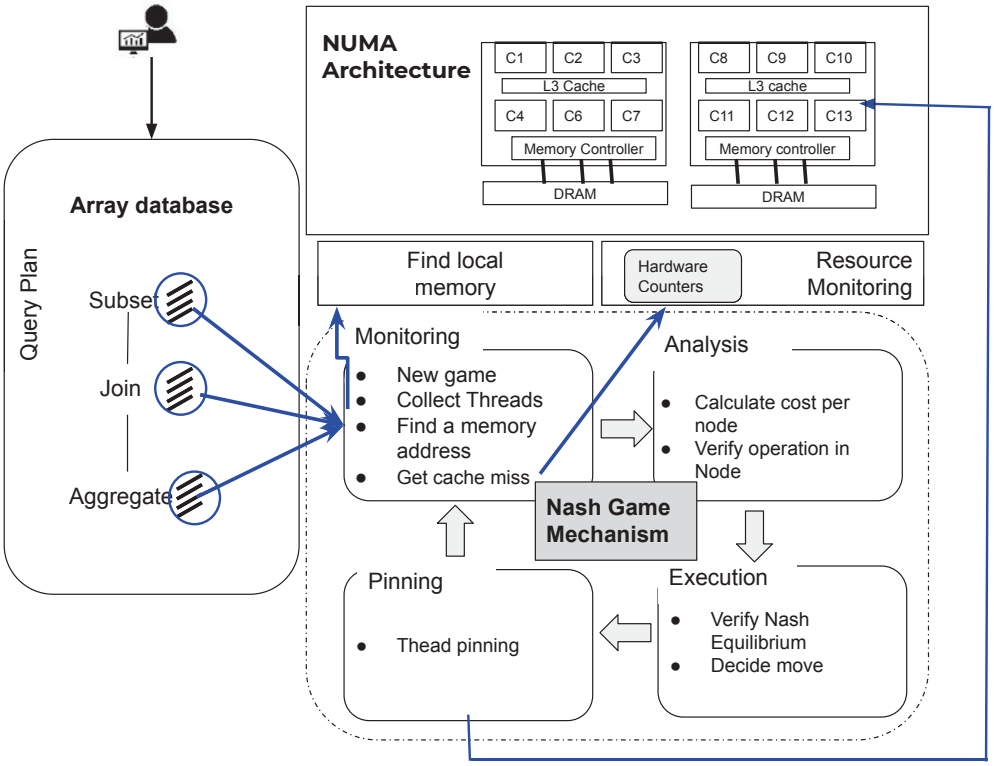


Figura 6.7: Visão geral do mecanismo *NasArray* proposto.

A Figura 6.7 apresenta uma visão do mecanismo baseado em equilíbrio de *Nash*. Quando um SGBD multidimensional é iniciado, o mecanismo começa um jogo. As informações coletadas no SGBD multidimensional são a identificação da *thread* e a operação que está executando. Sendo assim, o mecanismo age como um *middleware* que precisa de informações do SGBD multidimensional e, para isso, inclui algumas linhas de código no código-fonte do SGBD multidimensional. As *threads* são identificadas pelo seu TID e instanciadas como jogadores no jogo. Ao criar um jogador, verifica-se em que nó está o endereço de memória buscado. Na sequência, é realizada a coleta de informações do nó NUMA, que inclui o custo denotado pela equação já apresenta acima (Equação 6.1). O mecanismo verifica periodicamente o desempenho de uma *thread* e toma decisões sobre sua atribuição para a próxima iteração de escalonamento. Nesse caso, se seu custo é menor que o custo do nó NUMA alocado, a *thread* somente é migrada se existir um nó com menor custo.

É importante notar que a preferência do jogador é baseada sempre na função de custo: o jogador poderá se posicionar no local com menor custo. Entretanto, como se tratam de diferentes

padrões de acesso e reuso de memória, é necessário combinar algumas estratégias específicas de tais jogadores. Isso significa que, em cada padrão, a escolha da alocação da*thread* depende das operações que estão em um nó NUMA. Quando está em execução uma operação que possui alto reuso de dados e acesso à memória coalescente, a *thread* decidirá a sua alocação no nó NUMA em que o endereço de memória buscado está alocado. O mesmo comportamento é adotado por operações com alto reuso de dados e acessos não coalescentes. Entretanto, os dois padrões não se posicionam no mesmo local, ou seja, no mesmo nó.

Em oposição, a operação que possui baixo reuso de dados e acessos não coalescente vai se posicionar exatamente no nó que possui menor custo, mesmo não sendo o nó em que dados buscados por ela se encontram. Isso permite que as operações conflitantes com baixo reuso de dados e alto reuso de dados escolham nós diferentes. Entretanto, é considerado que nem sempre isso será possível se o SGBD multidimensional estiver executando muitas operações diferentes em paralelo. Assim, a função objetivo é a principal estratégia escolhida pelas *threads*, motivando o encontro de uma alocação em que seu *custo* seja o menor possível.

Além disso, o mecanismo assume que, durante a alocação de *threads*, elas não compartilham o núcleo de processamento. Por meio do *microbenchmark*, é possível observar que utilizar a tecnologia *Hyper-threading* não é uma boa política para as operações dos SGBDs multidimensionais. Quando é utilizada a estratégia *Compact*, o desempenho não melhora se comparado com as estratégias que posicionam as *threads* em núcleos individuais nos nós NUMA.

Por fim, para a implementação do mecanismo, foi utilizada a linguagem C. Esse mecanismo prescinde de informações do SGBD multidimensional: especificamente a operação em execução, a identificação da *thread* e os endereços buscados. O *NasArray* inicia com o SGBDs multidimensionais e coleta informações da topologia do *hardware*, utilizando a biblioteca *Portable Hardware Locality* (hwloc) [Broquedis et al. 2010]. Quando uma consulta é iniciada, o jogo começa a monitorar os contadores de *hardware*. A próxima seção apresenta a análise experimental do mecanismo *NasArray*.

6.3 ANÁLISE EXPERIMENTAL

Na análise experimental, foram utilizados dois SGBDs multidimensionais: SAVIME na versão (v.1.0) e SciDB na versão (v.19.11.5). Foram avaliados os desempenhos em três máquinas NUMA. Basicamente, a primeira máquina é utilizada nos experimentos, e utilizamos as outras máquinas para comparar o mecanismo em diferentes arquiteturas NUMA, é apontado o uso das outras duas máquinas para o SAVIME e SciDB.

A primeira máquina NUMA (aqui chamada de *NUMA-Skylake*) tem dois nós, cada nó com um *Intel Xeon Silver* 4114 (com microarquitetura *Skylake*). Cada soquete *Xeon* tem dez núcleos com *cache* L1 (I + D) privado (32 KB cada núcleo), um *cache* L2 privado (1 MB cada núcleo) e um *cache* L3 compartilhado (total de 14 MB por nó). Os dois nós NUMA são interconectados por um *link* QPI [Intel 2019] 4×, com largura de banda de 21, 5 GB/S. A máquina inclui 128 GB de memória principal DDR-4 e 14 TB de armazenamento em disco (a 15.000 rpm), executando o SO Ubuntu na versão 18.04.01 LTS para SAVIME e na versão 14.04.6 LTS para SciDB. As diferentes versões do SO foram usadas conforme a documentação do SGBD multidimensional, em um *kernel* Linux não-modificado, versão 4.15.0-121- genérico.

A segunda máquina (aqui chamada de *NUMA-SandyBridge*) também possui dois nós, cada nó com um *Intel Xeon E5*- 2630 (com microarquitetura *Sandy Bridge*). Cada nó *Xeon* tem seis núcleos físicos com *cache* L1 (I + D) privado (32 KB cada núcleo), um *cache* L2 privado (256 KB cada núcleo) e um *cache* L3 compartilhado (total de 15 MB por nó). Os dois nós NUMA são interconectados por um *link* QPI 4×, com largura de banda de 14, 4 GB/S. A máquina

inclui 48 GB de memória principal DDR-3 e 869 GB de armazenamento em disco (a 7500 rpm), executando o SO CentOs na versão 7.9.2009.

A terceira máquina (aqui chamada de *NUMA-NehalemEX*) possui quatro nós, cada nó com um *Intel Xeon- X*7550 (com microarquitetura *Nehalem*). Cada nó *Xeon* tem oito núcleos físicos com *cache* L1 (I + D) privado (32 KB cada núcleo), um *cache* L2 privado (256 KB cada núcleo) e um *cache* L3 compartilhado (total de 18 MB por nó). Os quatro nós NUMA são interconectados por um *link* QPI 4×, com largura de banda de $14,4$ GB/S. A máquina inclui 125 GB de memória principal DDR-3 e 5 TB de armazenamento em disco, executando o SO CentOs na versão 7.9.2009.

Para medir o desempenho do *hardware*, é utilizado o PCM [Willhalm Thomas 2012]. Em particular, a ferramenta *Intel* PCM fornece uma estimativa do consumo total de energia da memória principal e a energia consumida pela CPU. Nos experimentos, definiu-se o número máximo de *threads* disponíveis para cada execução de consulta corresponde ao número de núcleos físicos disponíveis. A carga de trabalho possui matrizes densas baseadas em dados do *benchmark* sísmico *HPC4e BSC* [Center 2016], usado no trabalho apresentado em [Lustosa e Porto 2019]. Os resultados são apresentados considerando a média de 10 execuções.

6.3.1 Impacto do número de *chunks*

Esta seção apresenta o impacto da alocação realizado pelo *NasArray* nos diferentes operadores de DB e na variação do número de *chunks*, com uma operação executando em simultâneo em um banco de dados de 1 GB, composto de duas matrizes densas para operação de junção e uma única matriz densa para agregação e *subarray*. Nos experimentos, foram utilizados os SGBDs multidimensionais SAVIME e SciDB, e a máquina *NUMA-Skylake*. Seu objetivo é medir o comportamento do *NasArray* em uma base de dados com diferentes quantidades de *chunks*. Os resultados são normalizados conforme os valores obtidos com 20 *chunks* do escalonador do SO.

A Figura 6.8 apresenta o resultado dos experimentos com 1 GB. No SAVIME, o NasArray alcançou um *speedup* de $1,64\times$ quando o número de *chunks* é igual ao número de núcleos da máquina. Como argumentado na Seção 6.1.3, uma quantidade de *chunks* menores aumenta o tamanho do *chunk* e, com isso, as *threads* não precisam trocar várias vezes os dados processados. Para o *NasArray*, essa descoberta facilitou ainda mais a coordenação do jogo, pois reduz a troca de contexto, o que auxilia a manutenção do equilíbrio do custo por um tempo maior – e, consequentemente, faz com que o *NasArray* não precise trocar as *threads* da sua alocação tão frequentemente. Isso resultou em uma redução de $2,46\times$ e $1,71\times$ dos acessos remotos, respectivamente, no SAVIME e SciDB. Além disso, os resultados sugerem que fixar a *thread* em um determinado núcleo melhora o aproveitamento da arquitetura NUMA.

As Figuras 6.9 e 6.10 mostram os resultados obtidos com as operações com alto reuso de dados – agregação e junção – normalizados pelo tamanho de *chunk* 20. Essas duas operações são posicionadas pelo *NasArray* exatamente no nó em que os dados estão localizados. No SAVIME, o mecanismo *NasArray* apresentou um *speedup* máximo de $1,38\times$ na junção e de $1,17\times$ na agregação. Além disso, o SO apresentou um bom desempenho com 20 *chunks*. Essas duas operações no SAVIME precisam percorrer todas as células das dimensões analisadas e, com muitos *chunks*, a dificuldade do *NasArray* em manter o equilíbrio é maior. Entretanto, o escalonamento do mecanismo otimiza a alocação e minimiza a movimentação de dados. Já no SciDB a junção e agregação obtêm o melhor resultado com 500 *chunks* chegando a $4,26$ na agregação e $2,14$ na junção. A agregação no SciDB possui um comportamento equilibrado visto que acessa exatamente os valores utilizados, assim a fixação das *threads* reduz os acessos remotos. Na junção o SciDB precisa passar por todas as dimensões ao fixar a *thread* exatamente

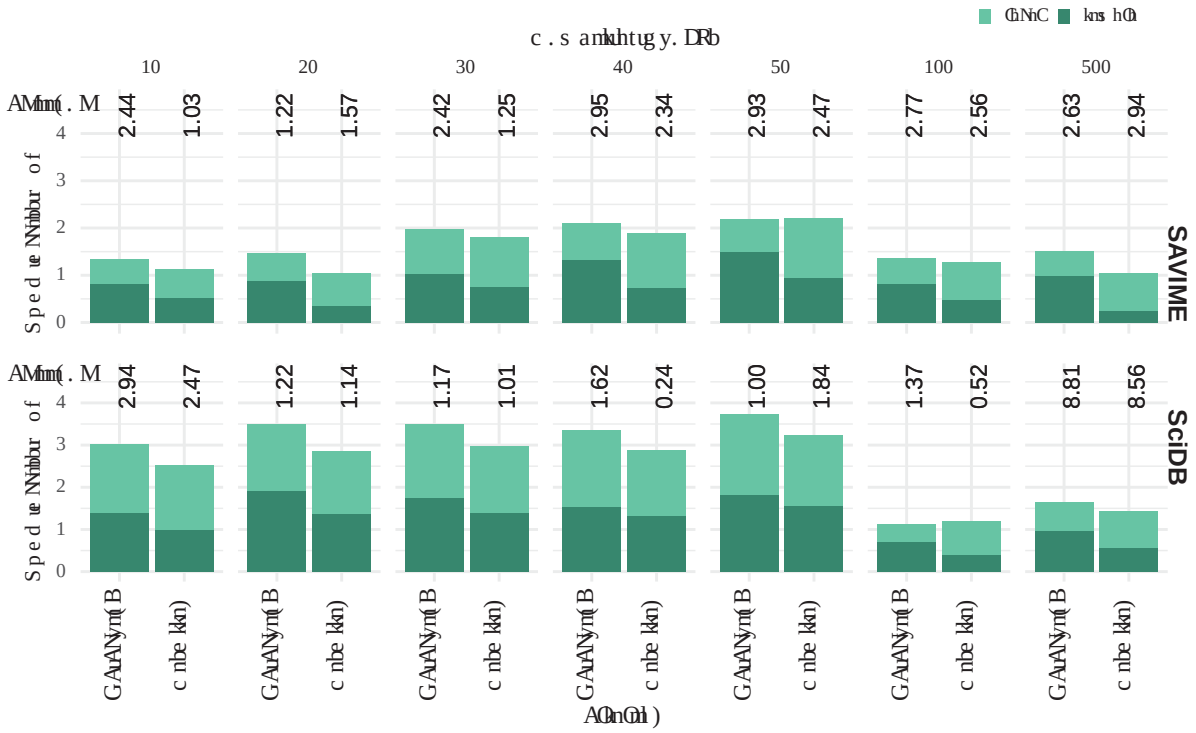


Figura 6.8: *Speedup* e quantidade de acessos à memória comparando o escalonador do SO com o mecanismo *NasArray*, variando a quantidade de *chunks*. Nos experimentos, foi utilizado um banco de dados de matriz de 1 GB na máquina *NUMA-Skylake* e a operação com baixo reuso de dados e acesso à memória coalescente (***subarray***). O número do eixo superior é o *speedup*. Valores normalizados com relação a 20 chuncks.

no nó NUMA com os dados melhora a utilização da memória reduzindo os acessos remotos e os acessos à memória.

6.3.2 Avaliando o comportamento dos operadores de banco de dados

Considerando que cada operação possui diferentes padrões de acesso à memória e a quantidade de dados ou *chunks* que são processados pode mudar em cada operação. Reduzir a quantidade de dados processados, pode-se reduzir ainda mais os acessos à memória remota caso o escalonamento das *threads* seja eficiente. Sendo assim, essa subseção avalia de forma isolada cada um dos operadores de banco de dados apresentados.

6.3.2.1 Avaliação da operação de subarray

Neste experimento, foi utilizada a operação de *subarray* com diferentes seletividades, processando todos os *chunks* (**M**) e apenas 20% dos *chunks* (**F**). A seletividade indica a porcentagem de dados que precisa ser filtrada para materializar a saída do *subarray*. Alta seletividade (**H** — neste experimento 70%) indica que se filtram mais dados e, consequentemente, menos dados são materializados para a saída em *DRAM*. Baixa seletividade (**L** — neste experimento 20%) indica o oposto.

A Figura 6.11 mostra os resultados alcançados para os SGBDs multidimensionais SAVIME e SciDB, executando na máquina *NUMA-Skylake*. Os gráficos são normalizados segundo os resultados do *baseline*, que é o escalonador do *OS* Linux (*OS Sched.*). Os resultados indicam que o mecanismo *NasArray* apresentou um bom comportamento, considerando a alta seletividade dos dados quando todos os *chunks* estão sendo processados, chegando a um *speedup*

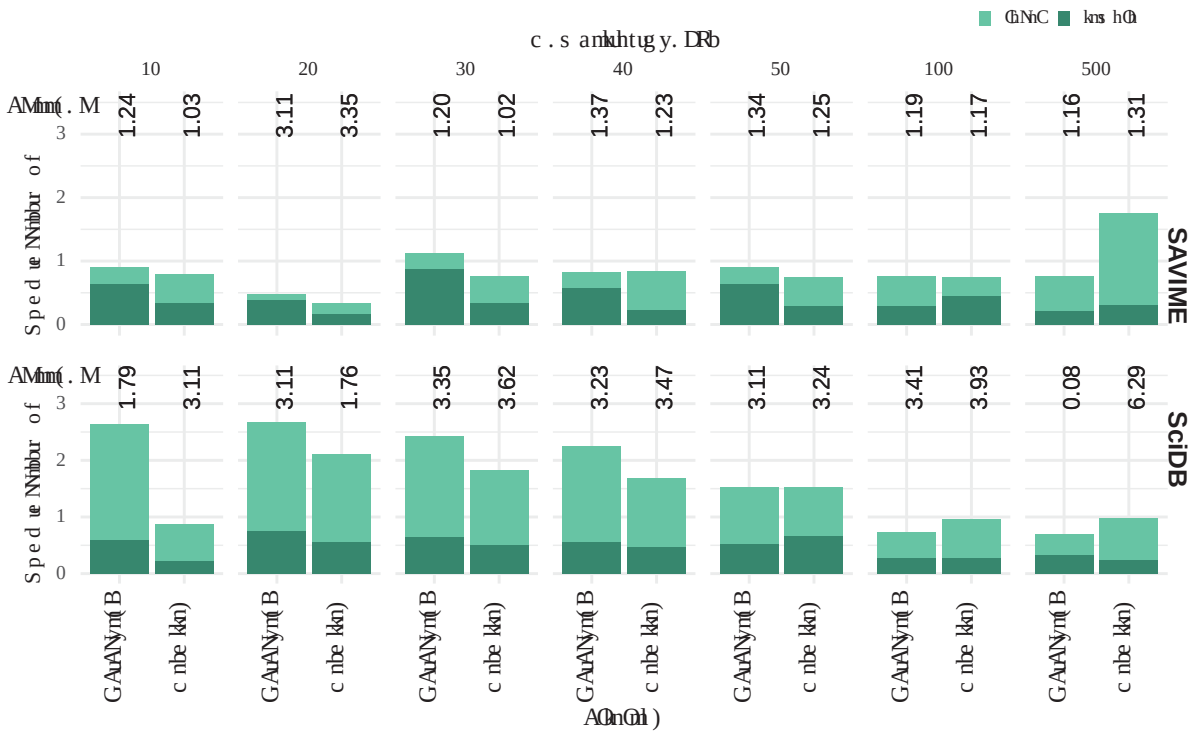


Figura 6.9: *Speedup* e quantidade de acessos à memória comparando o escalonador do SO com o mecanismo *NasArray*, variando a quantidade de *chunks*. Nos experimentos. foram utilizados um banco de dados de matriz de 1 GB na máquina *NUMA-Skylake*, e a operação com alto reuso de dados e acesso à memória não coalescente (**agregação**). O número do eixo superior é o *speedup*. Valores normalizados com relação a 20 chuncks.

de 2, 49× noSGBD multidimensional SAVIME. A explicação para isso é que, quando o SO é responsável pelo escalonamento das *threads*, ele acaba migrando-as entre os nós NUMA. Sendo assim, as *threads* que estão trabalhando juntas em um mesmo *chunk* acabam gerando acessos remotos à memória e faltas de dados na *cache*. Já o mecanismo *NasArray* fixa a *thread* no nó com o menor custo para esta operação durante o mapeamento. Ao adotá-lo, é possível verificar uma redução de 7× no acesso remoto, além de uma redução dos acessos à memória principal, indicando um bom uso da memória *cache*.

Outro ponto interessante é que os SGBDs multidimensionais são afetados de formas diferentes pela alocação de *thread*. Enquanto no SAVIME o *NasArray* apresenta os melhores resultados com alta seletividade, no SciDB os melhores resultados são com baixa seletividade. De fato, uma consulta com alta seletividade deve ser mais rápida porque retorna uma quantidade menor de dados. Entretanto, o SAVIME processa o *chunk* selecionando exatamente os dados no intervalo especificado. Já o SciDB precisa descompactar os dados e redistribuir entre as instâncias para, então, processá-los. Isso pode ter colaborado para o *NasArray* levar um tempo maior para encontrar o equilíbrio no SciDB e não ser tão eficiente quanto no SAVIME.

Como resultado, a alocação fixando as *threads* com alta seletividade mostra-se mais eficiente no SAVIME ao maximizar a localidade dos dados, minimizando a latência extra de acessos remotos. No SAVIME, a baixa seletividade acaba selecionando mais dados e atinge um rendimento mais baixo porque o *NasArray* tem mais trabalho para encontrar a alocação ideal, visto que o equilíbrio pode mudar se as faltas de dados na *cache* aumentarem. O SciDB precisa fazer o mesmo procedimento seja com alta ou baixa seletividade e, neste caso, com baixa seletividade os dados acabam sendo reaproveitados com a fixação da *thread*.

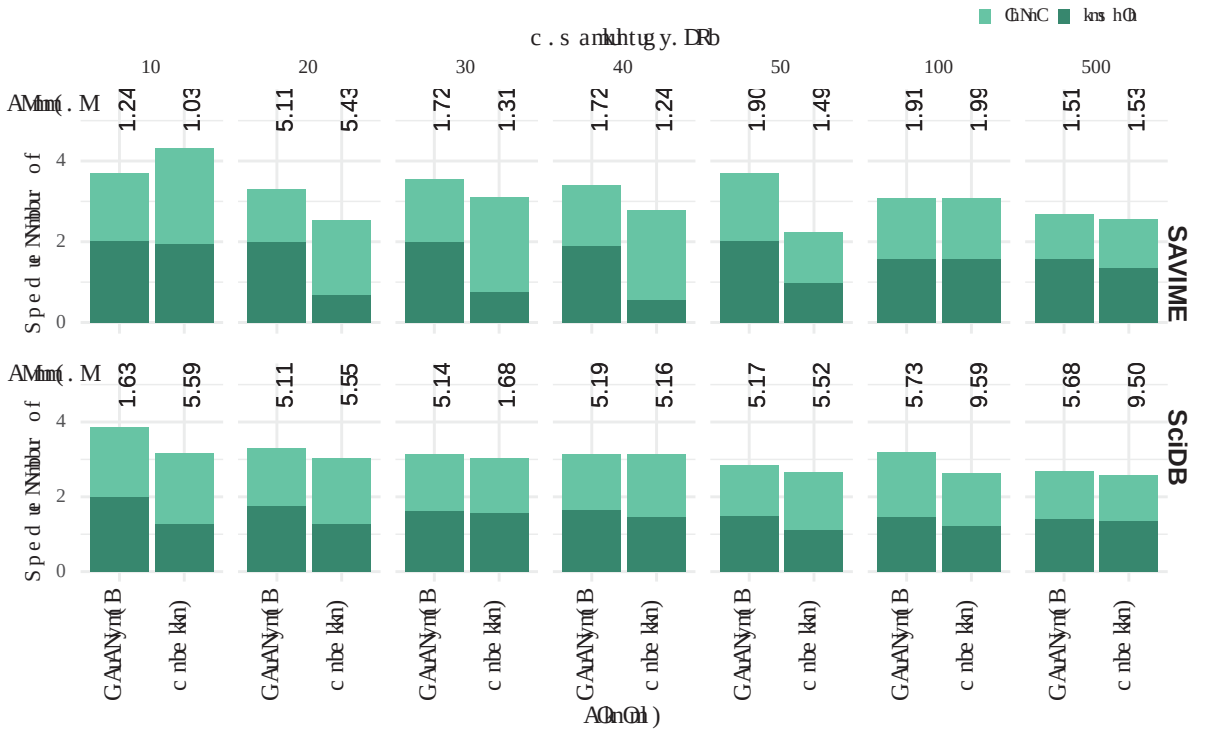


Figura 6.10: *Speedup* e quantidade de acessos à memória comparando o escalonador do SO com o mecanismo *NasArray*, variando a quantidade de *chunks*. Nos experimentos, foram utilizados um banco de dados de matriz de 1 GB na máquina*NUMA-Skylake*, e a operação com alto reuso de dados e acesso à memória coalescente (**junção**). O número do eixo superior é o *speedup*. Valores normalizados com relação a 20 chuncks.

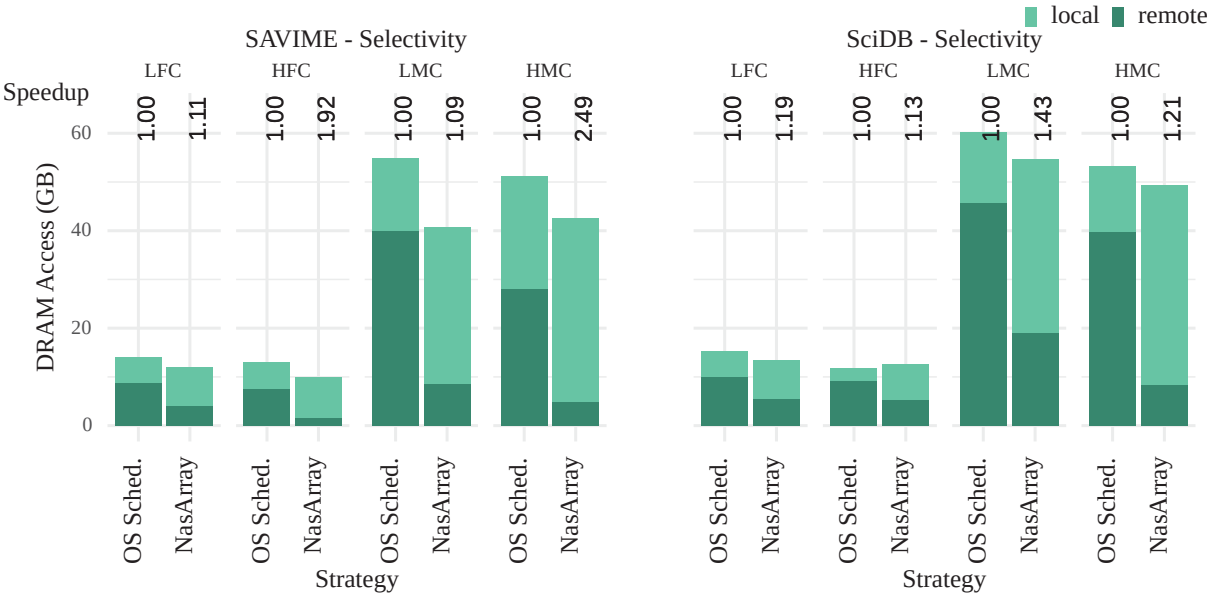


Figura 6.11: *Speedup* e quantidade de acessos à memória comparando o escalonador do SO com o mecanismo *NasArray*, variando a seletividade do **operador *subarray***. Os experimentos foram executados na *NUMA-Skylake*. O número no eixo superior é o *speedup*. Valores normalizados com relação a 20 chuncks.

6.3.2.2 Avaliação da operação de agregação

A operação de agregação tratada com o padrão de acesso n]ao coalescente à memória e com alto reuso de dados pode ser aplicada em diferentes dimensões. Neste experimento, foi

comparado o desempenho da agregação na primeira dimensão e na última. A Figura 6.12 mostra os resultados da agregação para o SAVIME e para o SciDB. Os resultados do *NasArray* foram semelhantes em ambos os SGBDs multidimensionais, alcançando um *speedup* de $1,30\times$ e uma redução de $1,50\times$ no acesso remoto.

No caso do SAVIME, os resultados da avaliação de cada *chunk* são armazenados em um *buffer* para serem reagrupados e gerar o resultado final. A alocação de *threads* fixadas em um determinado núcleo contribui para esses *buffers* manterem-se na memória de um determinado nó, pois são relativamente pequenos. É possível verificar que, quando a operação de agregação é realizada na última dimensão, o SciDB tem mais acessos à memória e o *speedup* alcançado é menor. O SciDB conduz agregação lendo cada célula necessária por meio de saltos entre elas. Quando a agregação é na última dimensão, o SciDB não consegue ler os dados contíguos na memória, levando a comportamentos de acesso dispersos. Essa característica faz com que o *NasArray* não apresente um desempenho maior. Como essa operação é tratada como alto reuso de dados e de acessos não coalescentes, quando os acessos ficam dispersos, o *NasArray* possivelmente demorou a encontrar o equilíbrio, já que as faltas de dados na *cache* devem aumentar consideravelmente e o *NasArray* verifica diferentes alocações para o equilíbrio.

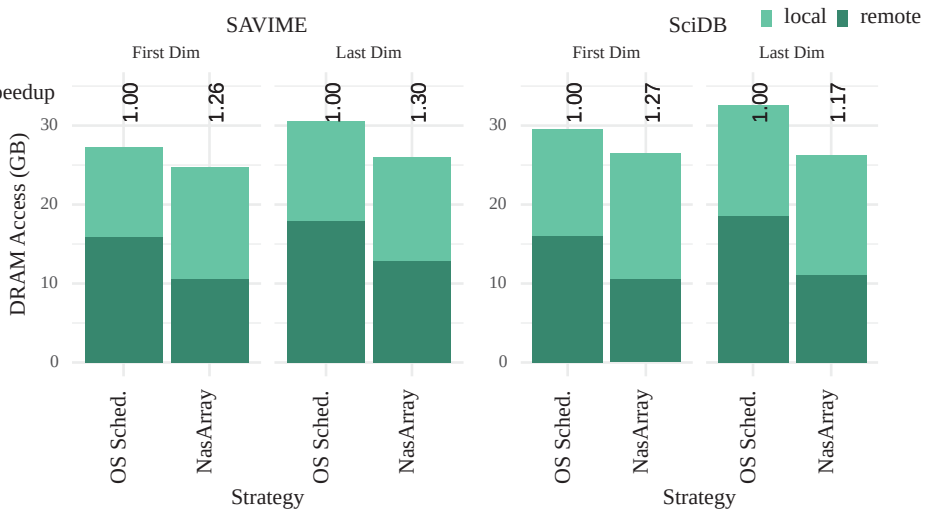


Figura 6.12: *Speedup* e quantidade de acessos à memória comparando o escalonador do SO com o mecanismo *NasArray*, aplicando a **operação de agregação** em diferentes dimensões. Nos experimentos, foram executados os padrões de acesso à memória em um banco de dados de matriz de 50 GB, na máquina *NUMA-Skylake*. O número do eixo superior é o *speedup*.

6.3.2.3 Avaliação da operação de junção

A operação de junção no SGBD multidimensional SAVIME pode ser aplicada a apenas algumas dimensões, ou seja, junção parcial. Já o SGBD multidimensional SciDB, na operação de junção, une necessariamente todas as dimensões da matriz. Considerando isso, foi avaliado o desempenho de junções parciais e de junção total para analisar o comportamento do escalonamento do mecanismo *NasArray*, considerando a base de dados utilizada nesta seção. No SAVIME, foram avaliadas a junção parcial e total. No SciDB, avaliou-se a junção total.

A Figura 6.13 apresenta os resultados de *speedup* e acessos à memória obtidos com as junções total e parcial no SAVIME e com a junção total no SciDB. Como a operação no SAVIME realiza um laço aninhado para combinar os pares das matrizes, quando a operação é realizada em apenas uma dimensão os acessos à memória são menores. O *NasArray* apresenta um desempenho

melhor se comparado ao escalonador do *OS*, chegando a um *speedup* de $1,39\times$. Como as *threads* trabalham juntas em um determinado *chunk*, o escalonamento de *threads* realizado pelo mecanismo *NasArray* evita que as *threads* realizem migrações desnecessárias e, com menos dados sendo processados, não é necessário analisar todas as dimensões como na junção total. Observando o SciDB, o *NasArray* apresentou um desempenho semelhante, chegando a $1,38\times$ de *speedup*. Além disso, os acessos à memória são menores. O SciDB consegue combinar os pares de células eficientemente porque as matrizes que compõe a base de dados possuem o mesmo número de *chunks* e tamanho. Assim, o mecanismo *NasArray* consegue reduzir os acessos remotos, alocando as *threads* exatamente no nó em que estão os dados e aproveitando o acesso contíguo à memória.

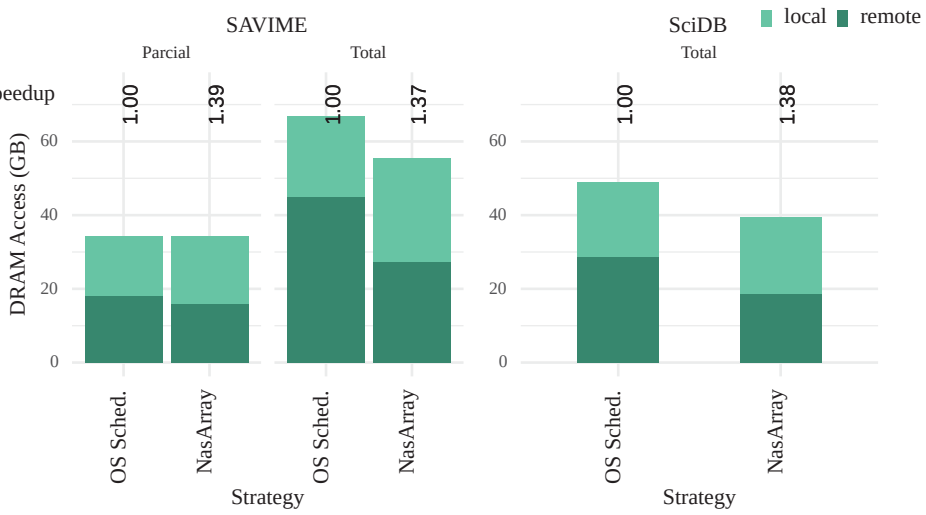


Figura 6.13: *Speedup* e quantidade de acessos à memória comparando o escalonador do SO com o mecanismo *NasArray*, aplicando a **operação de junção**. Nos experimentos, foram executados os padrões de acesso à memória em um banco de dados de matriz de 50 GB, na máquina *NUMA-Skylake*. O número do eixo superior é o *speedup*.

6.3.3 Eficiência energética do mecanismo *NasArray*

Para avaliar o consumo de energia, foi utilizada a métrica de EDP. Como é analisado o desempenho de uma aplicação *multi-threads*, essa métrica é um bom indicador para avaliar a relação entre energia e desempenho. O EDP determina o produto entre a energia e o tempo de execução. Assim, **quanto menor o valor, mais eficiente é o sistema**. Para o cálculo de energia, foram consideradas os consumos de energia do processador e da memória principal. Esses valores foram coletados utilizando a ferramenta PCM.

A Figura 6.14 mostra os resultados para uma base de dados de 1GB, variando a quantidade de *chunks*. É possível observar a mesma tendência apresentada nos resultados de *speedup* O SAVIME apresenta um EDP menor quando o número de *chunks* é próximo à quantidade de núcleos. Já o SciDB segue a tendência inversa. Outro ponto interessante é que o processamento tem consumo maior de energia quando comparado ao consumo da memória principal (DRAM). Em alguns casos, o processamento usando o mecanismo *NasArray* apresentou um custo maior de energia, que ficou evidenciado pelo valor de *EDP*: no SAVIME, na operação de *subarray* com 40 *chunks* e, no SciDB, na agregação com 20 *chunks*. Isso mostra que, em alguns casos, o *NasArray* precisou de alterações mais constantes no equilíbrio de alocação – mesmo apresentando um *speedup* melhor – o consumo de energia de processamento é maior

indicando uma analise maior para encontrar o equilíbrio. Além disso, a tendência de *speedup* reflete no consumo de energia apresentado pela *DRAM*.

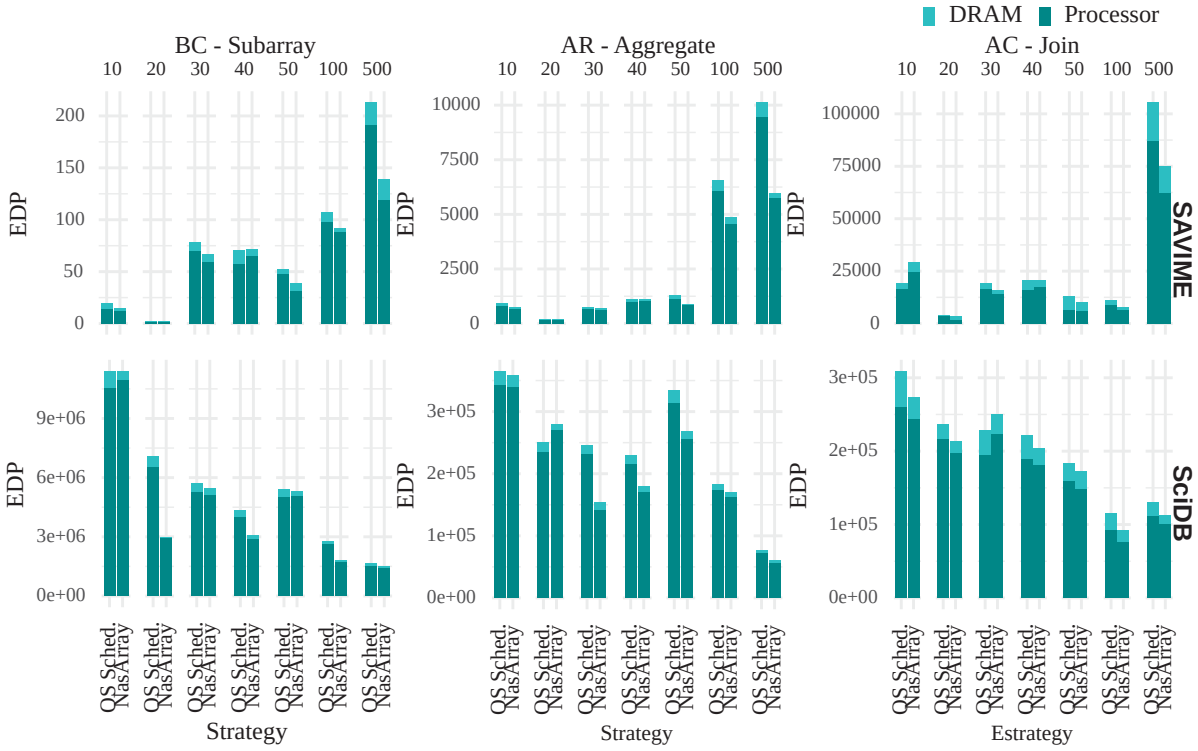


Figura 6.14: Avaliação de energia incluindo energia total (processamento + memória principal), usando a métrica *EDP* para uma base de dados de 1 GB, variando o número de *chunks*.

A Figura 6.15 apresenta os resultados de EDP para uma base de dados de 50 GB. Para a operação de *subarray*, apresenta os resultados com alta seletividade em parte dos *chunks* e em todos os *chunks*. Já a agregação foi realizada na primeira dimensão. O grau de melhoria observado no EDP apresenta sincronia com os resultados de *speedup*. Nesse caso, o mecanismo *NasArray* reduziu o EDP em todos os casos avaliados.

6.3.4 Desempenho do *NasArray* em arquiteturas *NUMA* diferentes

Para avaliar a efetividade da alocação de *threads* realizada pelo *NasArray* em arquiteturas NUMA distintas, foram realizados experimentos em diferentes máquinas NUMA, sendo elas: *NUMA-Skylake*, *NUMA-SandyBridge* e *NUMA-Nehalem*. Essas máquinas são de duas gerações diferentes de processadores Intel, ou seja, apresentam diferentes micro-arquiteturas e processos de litografia (miniaturização). Além disso, as três principais diferenças entre as máquinas são o número de núcleos, o tamanho do *cache* L3 e a diferença entre a latência da memória local e remota (fator NUMA). A máquina *NUMA-Skylake* tem 2× 10 núcleos compartilhando 14 MB de L3 e fator NUMA de 1, 36. Já a máquina *NUMA-SandyBridge* tem 2× 6 núcleos compartilhando 15 MB de L3 e fator NUMA de 1, 32. Por fim, a máquina *NUMA-Nehalem* possui 4× 8 núcleos compartilhando 18 MB de L3 e fator NUMA de 1, 35×. A *NUMA-Skylake* foi utilizada para experimentos com os dois SGBDs multidimensionais avaliados: *NUMA-SandyBridge* para o SAVIME e *NUMA-Nehalem* para o SciDB.

Para este experimento, foi utilizada uma base de dados de $50GB$, organizada em diferentes números de *chunks*. Os resultados foram normalizados pelo experimento com escalonador do SO com 100 *chunks*. Para o *subarray*, a consulta filtra metade dos dados, enquanto a agregação é

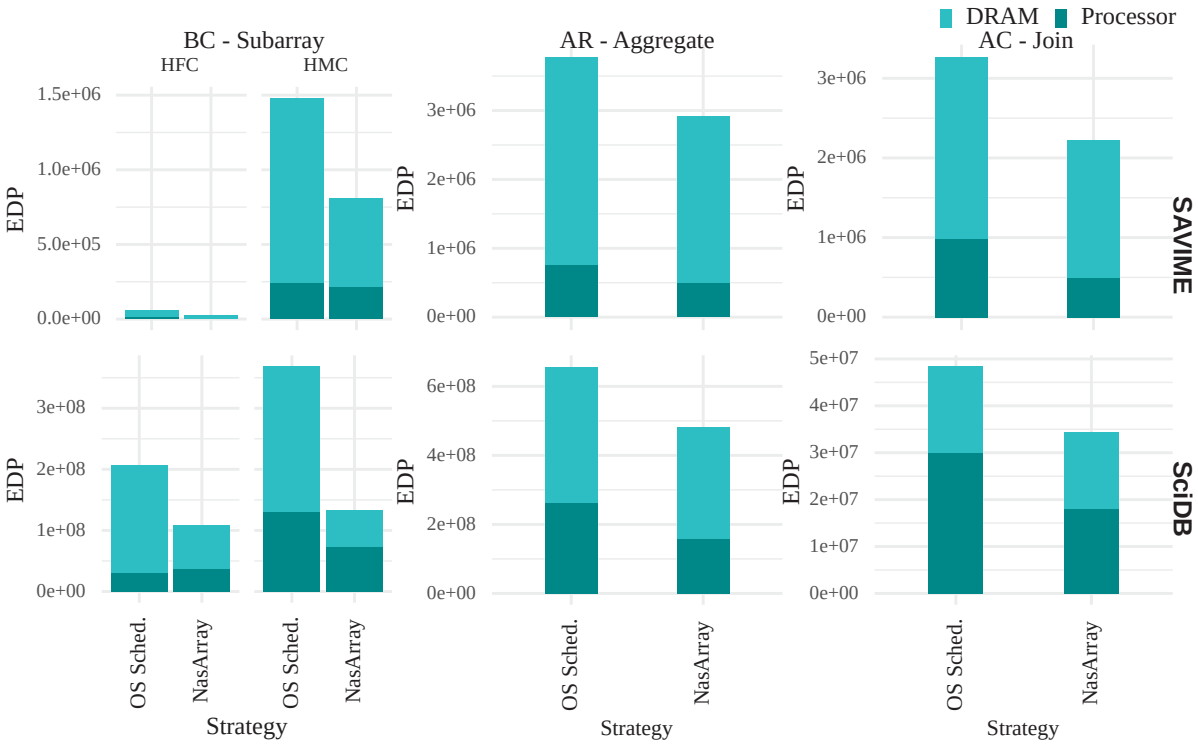


Figura 6.15: Avaliação de energia incluindo energia total (processamento + memória principal), usando a métrica *EDP*, para uma base de dados de 50 GB.

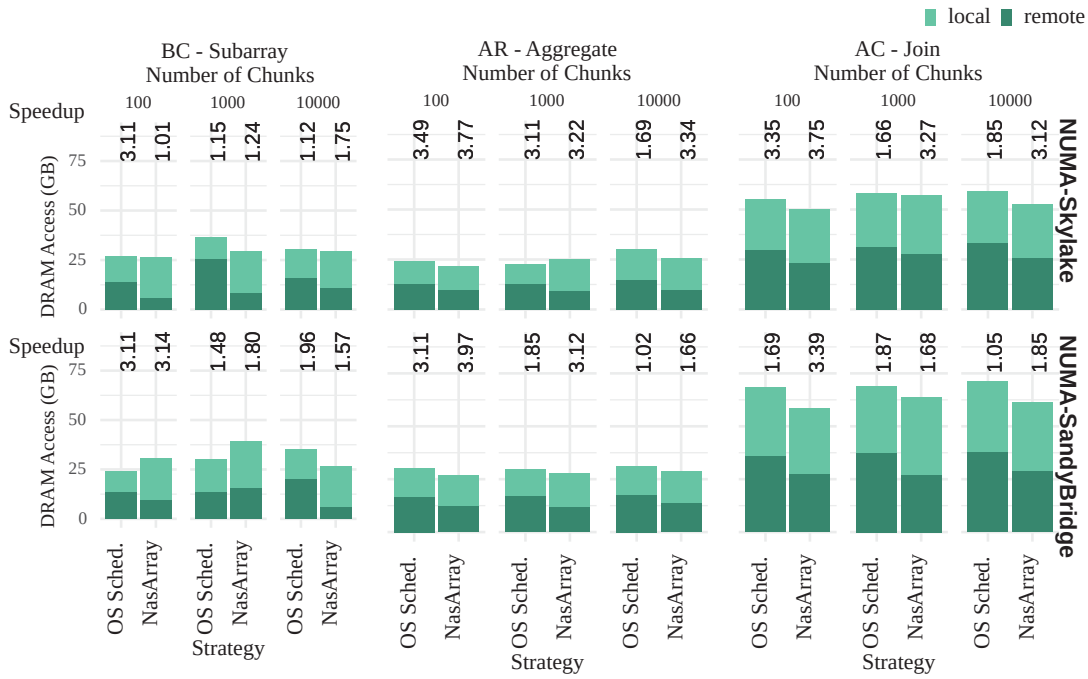


Figura 6.16: *Speedup* e quantidade de acessos à memória comparando o escalonador do SO com o mecanismo *NasArray*, alternando a quantidade de *chunks*. Nos experimentos, foram executados os diversos operadores em um banco de dados de matriz de 50 GB nas máquinas *NUMA-Skylake* e *NUMA-SandyBride*, com o SGBD multidimensional SAVIME. O número do eixo superior é o *speedup*.

aplicada na primeira dimensão e a junção é total. As Figuras 6.16 e 6.17 comparam os resultados da *NUMA-Skylake*, respectivamente, com as máquinas *NUMA-SandyBridge* e *NUMA-Nehalem*. De modo geral, os ganhos de *speedup* são mantidos nas diferentes arquiteturas, demonstrando

que o mecanismo *NasArray* consegue manter um bom desempenho em arquiteturas NUMA distintas. Mesmo lidando com latências, número de núcleos e nós NUMA diferentes, o *NasArray* reconhece a arquitetura e organiza o jogo baseado nas informações da máquina. Como esperado, o *NasArray* encontra uma boa alocação mesmo para máquinas com quatro nós NUMA, como a *NUMA-Nehalem*.

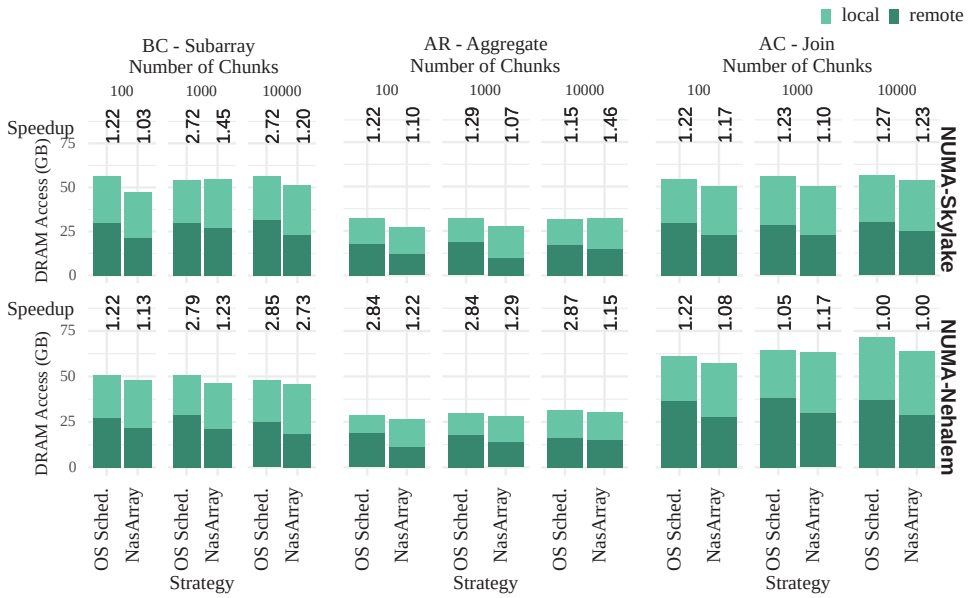


Figura 6.17: *Speedup* e quantidade de acessos à memória comparando o escalonador do SO com o mecanismo *NasArray*, alternando a quantidade de *chunks*. Nos experimentos, foram executados os padrões de acesso à memória em um banco de dados de matriz de 50 GB, nas máquinas *NUMA-Skylake* e *NUMA-Nehalem*, com o SGBD multidimensional SciDB. O número do eixo superior é o *speedup*.

6.4 CONSIDERAÇÕES FINAIS

O mecanismo *NasArray* apresentado neste capítulo mitigou a movimentação de dados e conseguiu melhorar o desempenho da execução de consultas analíticas em SGBDs multidimensionais utilizando a fixação da *thread* durante a alocação. Os resultados apresentados mostram melhorias de desempenho quando é realizada a fixação de *threads* em núcleos específicos. O mecanismo manteve o equilíbrio de desempenho durante o processamento de operações de consulta em matrizes multidimensionais. Além disso, o *NasArray* reduziu os acessos à memória e o consumo de energia em comparação com o escalonador do SO, explorando as informações de recursos computacionais e analisando os nós NUMA da perspectiva de um tomador de decisão dinâmico, aproveitando o potencial da fixação de *threads* e considerando a influência de falta de dados na *cache*.

7 CONCLUSÃO E TRABALHOS FUTUROS

Nessa tese, foram investigados os efeitos da arquitetura de computação NUMA na execução de consultas em SGBDs relacionais e de matriz multidimensional. A arquitetura de computação NUMA pode ser encontrada em *clusters* de *hardware* para implementação de banco de dados de grande escala, sendo projetada para evitar contenção no barramento de memória devido ao crescimento no número de núcleos de processamento. Entretanto, os SGBDs modernos, na sua maioria, não aproveitam o potencial da arquitetura NUMA. As abordagens de execução paralela de consultas não controlam a alocação das *threads* e entregam esta responsabilidade para o SO, que não possui conhecimento sobre a carga de trabalho de cada *threads* de execução. Portanto, foram apresentados dois mecanismos de posicionamento de *threads* que analisam as cargas de trabalho de consultas com o objetivo de mitigar a movimentação de dados da etapa de execução das consulta entre os nós de processamento NUMA.

Para aproveitar todo o potencial da arquitetura NUMA na etapa de execução de consultas, é importante analisar as características de acesso à memória das operações de consultas e analisar o comportamento de consumo de recursos computacionais por cada operação, visando otimizar a localidade e a alocação de *threads* e dados. Isso foi explorado de duas formas nesta tese: (1) controlando a alocação de núcleos disponíveis para o SO realizar o escalonamento e executando *threads* no nó NUMA em que os dados estão alocados ou em um nó próximo; (2) alocando as *threads* e controlando o escalonamento. Dessa forma, quando alocamos as *threads*, estamos de fato decidindo o mapeamento de *threads* e dados, de uma só vez, uma vez que sabemos que o SO faz o mapeamento *first-touch*.

Primeiro, foi apresentado um mecanismo de alocação de núcleos de processamento que controla o número de núcleos em que o SO escalona as *threads* de um SGBD relacional. Esse mecanismo utiliza informações de uso de recursos computacionais para alocar/remover núcleos disponíveis para o processamento de uma determinada carga de trabalho analítica. Além disso, o mecanismo utiliza a definição de Número Ótimo de Núcleos Local para determinar a quantidade de núcleos necessária para atender uma determinada carga de trabalho. O mecanismo reage a flutuações da carga de trabalho e disponibiliza os núcleos de forma dinâmica. Ademais, o mecanismo age como uma "caixa-preta", sem necessidade de alteração do código fonte de SGBDs relacionais. Na avaliação experimental, o controle da alocação dos núcleos pelo mecanismo mostrou um desempenho melhor quando comparado ao escalonamento do SO em termos de *speedup*, movimentação de dados e consumo de energia. O mecanismo pode melhorar o desempenho de cargas de trabalho analíticas, mantendo o controle dos estados de desempenho do SGBD relacional.

A seguir, foi apresentado um estudo sistemático de diferentes estratégias estáticas de alocação de *threads* para SGBDs multidimensionais. Foram analisadas todas os possíveis alocações para uma determinada arquitetura NUMA. Usando diferentes estratégias de alocação de *threads* nos SGBDs multidimensionais avaliados, foi mostrado como cada estratégia se comporta. Nos experimentos, observa-se que estratégias tradicionais têm um espaço de ganho de desempenho não-explorado, indicando a necessidade de um mecanismo inteligente que possua informações sobre as operações do banco de dados e da arquitetura NUMA. Sendo assim, os resultados apoiam que a arquitetura NUMA afeta severamente o desempenho da operação de *subarray* em um SGBD multidimensional.

As descobertas na análise dos efeitos da arquitetura NUMA em um SGBD multidimensional levaram ao desenvolvimento de um mecanismo que realiza o escalonamento de *threads* de

SGBDs multidimensionais, tentando posicionar as *threads* o mais próximo possível aos dados que são buscados. Considerando que a maioria dos cenários de execução de consultas em SGBDs multidimensionais envolve múltiplas *threads* colaborando ou competindo para realizar uma tarefa, apresentamos um mecanismo baseado na teoria dos jogos que traz uma modelagem e uma perceptiva de controle e alocação de *threads*. Nesta tese, foi possível analisar que a alocação de *threads* é um jogo multi-agente de tomada de decisão. As decisões são tomadas a partir de informações do padrão de acesso à memória das consultas e das faltas de dados na *cache* em cada nó NUMA. A avaliação experimental mostrou que o mecanismo posiciona as *threads* eficientemente na maioria das vezes, assumindo o escalonamento que antes era realizado pelo SO.

Por fim, considerando que, atualmente, a arquitetura NUMA é adotada em máquinas utilizadas por implementações de bancos de dados de larga escala, este projeto tem o potencial de melhorar o desempenho dos SGBDs onde estas implementações residem, mitigando a movimentação de dados e garantindo o uso eficiente dessa arquitetura.

7.1 TRABALHOS FUTUROS

Nesta tese, foram explorados dois tipos diferentes de SGBDs executando em sistemas multiprocessadores com arquitetura NUMA. O foco principal era melhorar o desempenho da etapa de execução de consulta destes sistemas, mitigando a movimentação de dados entre os nós de processamento NUMA. O trabalho desta tese mostra que existe um grande potencial para pesquisas futuras nessa área. Por exemplo, existe ainda a necessidade de analisar o impacto da arquitetura NUMA em cargas de trabalhos mistas (i.e., transacionais e analíticas em simultâneo) e adaptar os modelos desta tese para acomodar estas cargas de trabalho.

Nesse contexto, devem ser considerados o potencial da arquitetura NUMA e a consciência dos SGBDs sobre os recursos. Para isso, imagina-se um mecanismo dinâmico de análise da arquitetura que identifique a utilização dos recursos e decida entre fixação de *threads* ou controle de recursos como núcleos de CPU. Por meio do monitoramento de informações relevantes para a tomada de decisão, o mecanismo agiria em diferentes tipos de SGBDs reconhecendo a necessidade de cada modelo e tomando a decisão favorável para determinada carga de trabalho, baseado no conhecimento adquirido a partir de uma base de dados de aprendizagem.

Além disso, seria pertinente estudar mais operações de um SGBD multidimensional com objetivo de direcionar caracterizar outras operações nos padrões de acesso à memória. Os SGBDs multidimensionais estão sendo cada vez mais adotados para analisar diferentes cargas científicas. Assim, também seria relevante estudar o comportamento desses sistemas em diferentes arquiteturas heterogêneas que combinam CPU e GPU (*Graphics Processing Unit*) para analisar o potencial de desempenho em trabalhos futuros. Além disso, há pertinência de se analisar o comportamento dos SGBDs multidimensionais em arquiteturas emergentes, como as arquiteturas de *Processor-in-Memory* (PIM), com objetivo de explorar as melhorias oferecidas para o processamento de matrizes multidimensionais.

Por fim, é necessário estudar as melhorias necessárias para os SGBDs multidimensionais. Atualmente, existem diferentes pesquisam que analisam as melhorias para processamento de determinados tipos de dados científicos, porém poucas são direcionadas para analisar o potencial de soluções que considerem a arquitetura presente no *cluster* em que o SGBD multidimensional está executando.

REFERÊNCIAS

Abadi, D. J., Myers, D. S., DeWitt, D. J. e Madden, S. R. (2007). Materialization strategies in a column-oriented dbms. Em *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, páginas 466–475. IEEE.

Agrawal, S. R., Idicula, S., Raghavan, A., Vlachos, E., Govindaraju, V., Varadarajan, V., Balkesen, C., Giannikis, G., Roth, C., Agarwal, N. e Sedlar, E. (2017). A many-core architecture for in-memory data processing. Em *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17. ACM.

Ailamaki, A., Liarou, E., Tözün, P., Porobic, D. e Psaroudakis, I. (2017). Databases on modern hardware: How to stop underutilization and love multicores. *Synthesis Lectures on Data Management*, 9(1):1–113.

Albutiu, M.-C., Kemper, A. e Neumann, T. (2012). Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10).

Balkesen, C., Alonso, G., Teubner, J. e Özsu, M. T. (2013). Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1).

Barrera, I. S., Moretó, M., Labarta, J., Valero, M. e Casas, M. (2018). Reducing data movement on large shared memory systems by exploiting computation dependencies. *ACM International Conference on Supercomputing*.

Baumann, P., Furtado, P., Ritsch, R. e Widmann, N. (1997). The rasdaman approach to multidimensional database management. Em *ACM Symp. on Applied Computing*, páginas 166–173.

Baumann, P. e Holsten, S. (2011). A comparative analysis of array models for databases. Em *Database theory and application, bio-science and bio-technology*, páginas 80–89. Springer.

Baumann, P., Misev, D., Merticariu, V. e Huu, B. P. (2021). Array databases: concepts, standards, implementations. *Journal of Big Data*, 8(1):1–61.

Bellamkonda, S., Li, H., Jagtap, U., Zhu, Y., Liang, V. e Cruanes, T. (2013). Adaptive and big data scale parallel execution in oracle. *PVLDB*, 6(11):1102–1113.

Blanas, S., Wu, K., Byna, S., Dong, B. e Shoshani, A. (2014). Parallel data analysis directly on scientific file formats. Em *Proc. ACM SIGMOD Int. Conf. on Manag. of Data*, páginas 385–396.

Boncz, P., Neumann, T. e Erling, O. (2014). TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. Em *TPCTC*.

Boncz, P., Zukowski, M. e Nes, N. (2005). Monetdb/x100: Hyper-pipelining query execution. Em *In CIDR*.

Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S. e Namyst, R. (2010). hwloc: A generic framework for managing hardware affinities in hpc applications. Em *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, páginas 180–186. IEEE.

Brown, P. G. (2010). Overview of scidb: large scale array storage, processing and analysis. Em *Proc. ACM SIGMOD Int. Conf. on Manag. of data*, páginas 963–968.

Center, B. S. (2016). Hpc4e seismic test suite to increase the space of development of new modelling.

Chasparis, G. C., Rossbory, M. e Janjic, V. (2017). Efficient dynamic pinning of parallelized applications by reinforcement learning with applications. Em *Euro-Par: Parallel Processing*, páginas 164–176.

Chasparis, G. C., Rossbory, M., Janjic, V. e Hammond, K. (2019). Learning-based dynamic pinning of parallelized applications in many-core systems. Em *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, páginas 1–8. IEEE.

Chiang, M.-L., Tu, S.-W., Su, W.-L. e Lin, C.-W. (2018). Enhancing inter-node process migration for load balancing on linux-based numa multicore systems. Em *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, páginas 394–399.

Codd, E. F. (1985). Does your dbms run by the rules. *ComputerWorld*, 21.

Committee, O. (2013). Openmp 4.0 complete specifications. Relatório técnico, OpenMP Committee Technical Report.

Cruz, E. H., Diener, M., Alves, M. A., Pilla, L. L. e Navaux, P. O. (2016a). Lapt: A locality-aware page table for thread and data mapping. *Parallel Computing*, 54:59–71. 26th International Symposium on Computer Architecture and High Performance Computing.

Cruz, E. H., Diener, M., Pilla, L. L. e Navaux, P. O. (2021). Online thread and data mapping using a sharing-aware memory management unit. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 5(4):1–28.

Cruz, E. H. M., Diener, M., Pilla, L. L. e Navaux, P. O. A. (2016b). Hardware-assisted thread and data mapping in hierarchical multicore architectures. *ACM Trans. Archit. Code Optim.*, 13(3):28:1–28:28.

da Cruz, E. H. M., Alves, M. A. Z., Carissimi, A., Navaux, P. O. A., Ribeiro, C. P. e Méhaut, J. (2012). Memory-aware thread and data mapping for hierarchical multi-core platforms. *IJNC*, 2(1):97–116.

Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V. e Roth, M. (2013). Traffic management: A holistic approach to memory placement on numa systems. *SIGARCH Comput. Archit. News*, 41(1):381–394.

Denoyelle, N., Goglin, B., Jeannot, E. e Ropars, T. (2019). Data and thread placement in numa architectures: A statistical learning approach. Em *ICPP*, páginas 1–10.

Desel, J. e Reisig, W. (2015). The concepts of petri nets. *Software and System Modeling*, 2.

Di Gennaro, I., Pellegrini, A. e Quaglia, F. (2016). Os-based numa optimization: Tackling the case of truly multi-thread applications with non-partitioned virtual page accesses. Em *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, páginas 291–300.

Diener, M., Cruz, E. H. M., Alves, M. A. Z., Borin, E. e Navaux, P. O. A. (2017). Optimizing memory affinity with a hybrid compiler/os approach. Em *Proceedings of the Computing Frontiers Conference*, CF'17, página 221–229. Association for Computing Machinery.

Diener, M., Cruz, E. H. M., Alves, M. A. Z., Navaux, P. O. A., Busse, A. e Heiss, H.-U. (2016). Kernel-based thread and data mapping for improved memory affinity. *IEEE Trans. Parallel Distrib. Syst.*, 27(9):2653–2666.

Diener, M., Cruz, E. H. M. e Navaux, P. O. A. (2015). Locality vs. balance: Exploring data mapping policies on numa systems. Em *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, páginas 9–16.

Diener, M., Cruz, E. H. M., Navaux, P. O. A., Busse, A. e Heiß, H.-U. (2014). kmaf: Automatic kernel-level management of thread and data affinity. Em *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, páginas 277–288.

Dominico, S., de Almeida, E. C. e Meira, J. A. (2017). A petrinet mechanism for olap in numa. Em *Thirteenth International Workshop on Data Management on New Hardware*, DAMON '17, New York, NY, USA. Association for Computing Machinery.

Drepper, U. (2007). What every programmer should know about memory.

Dreseler, M., Kissinger, T., Djürken, T., Lübke, E., Uflacker, M., Habich, D., Plattner, H. e Lehner, W. (2017). Hardware-accelerated memory operations on large-scale numa systems. Em *ADMS@VLDB*.

Elmasri, R. e Navathe, S. B. (2000). *Fundamentals of Database Systems, 3rd Edition*. Addison-Wesley-Longman.

Elmasri, R., Navathe, S. B., Pinheiro, M. G. et al. (2005). *Sistemas de banco de dados*. Pearson Addison Wesley São Paulo.

Gaud, F., Lepers, B., Funston, J., Dashti, M., Fedorova, A., Quéma, V., Lachaize, R. e Roth, M. (2015). Challenges of memory management on modern numa systems. *Communications of the ACM*, 58(12):59–66.

Gawade, M., Kersten, M. e Simitsis, A. (2016). Multi-core column-store parallelization under concurrent workload. Em *DaMoN*.

Gawade, M. e Kersten, M. L. (2013). Tomograph: highlighting query parallelism in a multi-core system. Em *DBTest*, páginas 3:1–3:6.

Gawade, M. e Kersten, M. L. (2015). Numa obliviousness through memory mapping. Em *DaMoN*.

Gerhardt, L., Faham, C. e Yao, Y. (2015). Accelerating scientific analysis with scidb. *Journal of Physics: Conf. Series*, 664(7).

Giceva, J., Alonso, G., Roscoe, T. e Harris, T. (2014). Deployment of query plans on multicores. *Proc. VLDB Endow.*, páginas 233–244.

Giceva, J., ioan Salomie, T., Schüpbach, A., Alonso, G. e Roscoe, T. (2013). Cod: Database / operating system co-design. *CIDR13*.

Giceva, J., Zellweger, G., Alonso, G. e Rosco, T. (2016). Customized os support for data-processing. Em *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16. ACM.

Graefe, G. (1990). Encapsulation of parallelism in the volcano query processing system. Em *SIGMOD.*, páginas 102–111.

Group, B. (2016). The barrelfish operational systems. `http://www.barrelfish.org/documentation.html`. Acessado em 30/07/2018.

Hahn, K., Reiner, B. e Höfling, G. (2003). Parallel query support for multidimensional data: Intra-object parallelism. Em Mařík, V., Retschitzegger, W. e Štěpánková, O., editores, *Database and Expert Systems Applications*, páginas 212–222, Berlin, Heidelberg. Springer Berlin Heidelberg.

He, X. (1996). A formal definition of hierarchical predicate transition nets. Em *Application and Theory of Petri Nets*.

Hellerstein, J. M., Stonebraker, M. e Hamilton, J. (2007). *Architecture of a database system*. Now Publishers Inc.

Hey, T., Tansley, S. e Tolle, K. (2009). *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research.

Holst, A. (2021). Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025.

Intel (2019). Maximizing multicore processor performance.

Kemper, A. e Neumann, T. (2011). Hyper: A hybrid oltp amp;olap main memory database system based on virtual memory snapshots. Em *2011 IEEE 27th International Conference on Data Engineering*, páginas 195–206.

Kepe, T. R. (2019). *The Design and Implementation of Query Execution in Modern Processing-in-Memory Hardware*. Tese de doutorado, UFPR - Universidade Federal do Paraná, Curitiba - Brasil. 115 pgs.

Kiefer, T., Schlegel, B. e Lehner, W. (2013). Experimental evaluation of numa effects on database management systems. Em *BTW*.

Kim, S., Sohn, S. G., Kim, T., Yu, J., Kim, B. e Moon, B. (2016). Selective scan for filter operator of scidb. Em *Proceedings of the 28th Int. Conf. on Scientific and Statistical Database Manag.*, páginas 1–4.

Kissinger, T., Kiefer, T., Schlegel, B., Habich, D., Molka, D. e Lehner, W. (2014). ERIS: A NUMA-aware in-memory storage engine for analytical workload. Em *ADMS*.

Larson, P., Hanson, E. N. e Price, S. L. (2012). Columnar storage in SQL server 2012. *IEEE Data Eng. Bull.*, 35(1):15–20.

Leis, V., Boncz, P. A., Kemper, A. e Neumann, T. (2014). Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. Em *SIGMOD*, páginas 743–754.

Lepers, B., Quéma, V. e Fedorova, A. (2015). Thread and memory placement on numa systems: Asymmetry matters. Em *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, páginas 277–289.

Li, Y., Pandis, I., Mueller, R., Raman, V. e Lohman, G. M. (2013). NUMA-aware algorithms: the case of data shuffling. Em *CIDR*.

Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V. e Fedorova, A. (2016a). The linux scheduler: a decade of wasted cores. Em *Proceedings of the Eleventh European Conference on Computer Systems*, páginas 1–16.

Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V. e Fedorova, A. (2016b). Your cores are slacking off—or why os scheduling is a hard problem. *login Usenix Mag.*, 41(4).

Lustosa, H., Lemus, N., Porto, F. e Valduriez, P. (2017). Tars: An array model with rich semantics for multidimensional data. Em *Proc. of the ER Forum 2017 and the ER 2017 Demo Track*, páginas 114–127.

Lustosa, H. e Porto, F. (2019). SAVIME: A multidimensional system for the analysis and visualization of simulation data. *CoRR*, abs/1903.02949.

Lustosa, H., Porto, F., Blanco, P. e Valduriez, P. (2016). Database system support of simulation data. *Proc. VLDB Endow.*, 9(13):1329–1340.

Lustosa, H. L. S. (2020). *SAVIME: Enabling Declarative Array Processing In Memory*. Tese de doutorado, Laboratório Nacional de Computação Científica, Petrópolis - Brasil. 100 pgs.

Manegold, S., Boncz, P. e Kersten, M. L. (2002). Generic database cost models for hierarchical memory systems. Em *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, páginas 191–202. Elsevier.

Marathe, A. P. e Salem, K. (1999). Query processing techniques for arrays. Em *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, página 323–334, New York, NY, USA. Association for Computing Machinery.

Maziero, C. (2019). Sistemas operacionais: Conceitos e mecanismos. `http://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=socm:socm-livro.pdf`. Acessado em 30/04/2021.

Memarzia, P., Ray, S. e Bhavsar, V. C. (2020). The art of efficient in-memory query processing on NUMA systems: a systematic approach. Em *ICDE*, páginas 781–792.

Microsoft Docs (2018). Soft-numa (sql server). `https://docs.microsoft.com/pt-br/sql/database-engine/configure-windows/soft-numa-sql-server?view=sql-server-2017`. Acessado em 13/08/2018.

Minhas, U. F., Liu, R., Aboulnaga, A., Salem, K., Ng, J. e Robertson, S. (2012). Elastic scale-out for partition-based database systems. Em *ICDEW12*.

Montgomery, D. (2007). *Introduction to Statistical Quality Control*. Cram 101 textbook outlines. Academic internet publishers publication.

Müller, S. e Plattner, H. (2012). An in-depth analysis of data aggregation cost factors in a columnar in-memory database. Em *Proceedings of the fifteenth international workshop on Data warehousing and OLAP*, páginas 65–72.

Nash, J. (1951). Non-cooperative games. *Annals of mathematics*, 54:286–295.

Neumann, T. (2011). Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*

Oliveira, R. S., Carissimi, A. S. e Toscani, S. S. (2009). *Sistemas Operacionais-Vol. 11: Série Livros Didáticos Informática UFRGS*. Bookman Editora.

Ozisikyilmaz, B., Narayanan, R., Zambreno, J., Memik, G. e Choudhary, A. (2006). An architectural characterization study of data mining and bioinformatics workloads. Em *2006 IEEE International Symposium on Workload Characterization*, páginas 61–70. IEEE.

Özsu, M. T. e Valduriez, P. (1996). Distributed and parallel database systems. *ACM Computing Surveys (CSUR)*, páginas 125–128.

Ozturk, O., Orhan, U., Ding, W., Yedlapalli, P. e Kandemir, M. T. (2016). Cache hierarchy-aware query mapping on emerging multicore architectures. *IEEE Trans. on Computers*, páginas 403–415.

Papadopoulos, S., Datta, K., Madden, S. e Mattson, T. (2016). The tiledb array data storage manager. *Proc. VLDB Endow.*, 10(4):349–360.

Pavlo, A. (2017). Query processing. `https://15445.courses.cs.cmu.edu/fall2017/slides/10-queryprocessing.pdf`. Acessado em 20/11/2018.

Pilla, L. L., Pousa Ribeiro, C., Cordeiro, D., Bhatele, A., Navaux, P. O. A., Mehaut, J.-F. e Kalé, L. V. (2011). Improving Parallel System Performance with a NUMA-aware Load Balancer. Research report, Inria.

Pilla, L. L., Ribeiro, C. P., Cordeiro, D., Mei, C., Bhatele, A., Navaux, P. O. A., Broquedis, F., Méhaut, J. F. e Kale, L. V. (2012). A hierarchical approach for load balancing on parallel multi-core systems. Em *2012 41st International Conference on Parallel Processing*, páginas 118–127.

Popov, M., Jimborean, A. e Black-Schaffer, D. (2019). Efficient thread/page/parallelism autotuning for numa systems. Em *Int. Conf. on Supercomputing*, páginas 342–353.

Porobic, D., Liarou, E., Tözün, P. e Ailamaki, A. (2014). Atrapos: Adaptive transaction processing on hardware islands. Em *2014 IEEE 30th International Conference on Data Engineering*, páginas 688–699.

Porobic, D., Pandis, I., Branco, M., Tözün, P. e Ailamaki, A. (2012). OLTP on hardware islands. *PVLDB*, 11.

Psaroudakis, I., Scheuer, T., May, N., Sellami, A. e Ailamaki, A. (2015). Scaling up concurrent main-memory column-store scans: Towards adaptive NUMA-aware data and task placement. *PVLDB*, 12.

Psaroudakis, I., Scheuer, T., May, N., Sellami, A. e Ailamaki, A. (2016). Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *PVLDB*, 2.

Raman, V., Attaluri, G. K., Barber, R., Chainani, N., Kalmuk, D., KulandaiSamy, V., Leenstra, J., Lightstone, S., Liu, S., Lohman, G. M., Malkemus, T., Müller, R., Pandis, I., Schiefer, B., Sharpe, D., Sidle, R., Storm, A. J. e Zhang, L. (2013). Db2 with blu acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091.

Ray, S., Higgins, C., Anupindi, V. e Gautam, S. (2020). Enabling numa-aware main memory spatial join processing: An experimental study. Em *ADMS@ VLDB*.

Sergey, B., Sergey, Z., Mohammad, D. e Alexandra, F. (2011). A case for numa-aware contention management on multicore systems. Em *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, páginas 1–1, Berkeley, CA, USA.

Serpa, M. S., Krause, A. M., Cruz, E. H., Navaux, P. O. A., Pasin, M. e Felber, P. (2018). Optimizing machine learning algorithms on multi-core and many-core architectures using thread and data mapping. Em *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, páginas 329–333. IEEE.

SGI (2011). Sgi uv revolutionary x86 performance and scalability. `http://www.risusbilisim.com/AltixUV_datasheet.pdf`.

Silberschatz, A., Galvin, P. e Gagne, G. (2014). *Fundamentos de Sistemas Operacionais*. LTC.

Silberschatz, A., Galvin, P. B. e Gagne, G. (2001). *Sistemas operacionais: conceitos e aplicações*. Campus.

Silberschatz, A., Korth, H. e Sudarshan, S. (2006). *Sistema de banco de dados*. Elsevier.

Silberschatz, A., Korth, H. F. e Sudarshan, S. (2010). *Database system concepts*. McGraw-Hill, 6 edition.

Soroush, E., Balazinska, M. e Wang, D. (2011). Arraystore: a storage manager for complex parallel array processing. Em *Proc. ACM SIGMOD Int. Conf. on Manag. of data*, páginas 253–264.

Stonebraker, M. (1981). Operating system support for database management. *Commun. ACM*, 24(7):412–418.

Tanenbaum, A. S. (2004). *Sistemas operacionais modernos*, volume 2. Pearson Prentice-Hall.

Treibig, J., Hager, G. e Wellein, G. (2010). Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. Em *PSTI*.

Ullman, J. D. (1975). Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393.

Vikranth, B., Wankar, R. e Rao, C. R. (2013). Topology aware task stealing for on-chip numa multi-core processors. *Procedia Computer Science*, 18:379–388.

Wang, Q. e Lee, B. C. (2016). Modeling communication costs in blade servers. *SIGOPS Oper. Syst. Rev.*, 49(2):75–79.

Wang, W., Davidson, J. W. e Soffa, M. L. (2016). Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. Em *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, páginas 419–431.

White, P. (2013). Parallel execution plans – branches and threads. `https://sqlperformance.com/2013/10/sql-plan/parallel-plans-branches-threads`. Acessado em 20/07/2018.

Willhalm Thomas, Dementiev Roman, F. P. (2012). Intel performance counter monitor.

Zeuch, S. e Freytag, J.-C. (2015). Selection on modern cpus. Em *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, páginas 1–8.

Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L. e Zhang, M. (2015). In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948.

Zhang, Y., Kersten, M. e Manegold, S. (2013). Sciql: array data processing inside an rdbms. Em *Proc. ACM SIGMOD Int. Conf. on Manag. of Data*, páginas 1049–1052.

Zheng, G., Kale, L., Bohm, E., Bhatele, A., Mei, C., Gioachin, F., Venkataraman, R., Sun, Y., Acun, B., White, S. e Galvez, J. (2017). Parallel programming with migratable objects. `http://charm.cs.illinois.edu/research/charm`. Acessado em 10/07/2018.