PROPOSTA E AVALIAÇÃO DE MECANISMOS DE *SOFTWARE* DE MEMÓRIA COMPARTILHADA DISTRIBUÍDA PARA E/S PARALELA

Carla Osthoff Ferreira de Barros

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Cláudio Luis de Amorim, Ph.D.

Prof. Marta L. Q. Mattoso, D.Sc.

Prof. Inês de Castro Dutra, Ph.D.

Prof. Sergio Lifschitz, D.Sc.

Prof. Wagner Meira Júnior, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2000

OSTHOFF, CARLA FERREIRA DE BARROS

Proposta e Avaliação de Mecanismos de *Software* de Memória Compartilhada Distribuída em E/S Paralela.[Rio de Janeiro] 2000

XV, 124 p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computacão, 2000)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Sistemas Operacionais

2 - Memória Compartilhada Distribuída

3 - Sistemas Gerentes de Banco de Dados Paralelos

4 - Sistemas de Entrada e Saída de Dados Paralelos

I. COPPE/UFRJ II. Título (série)

Aos meus pais Aída e Linton

Agradecimentos

Agradeço:

à Cláudio Luis de Amorim, meu orientador, por tudo, pelas inúmeras sugestões ao longo do desenvolvimento da minha tese, pelas incansáveis revisões do texto, pela confiança, estímulo e apoio incondicional nas horas em que mais precisei;

aos órgãos financiadores Finep, CAPES e CNPq, que deram suporte a realização desse trabalho;

ao Laboratório Nacional de Computação Científica, e em particular, aos diretores Antônio César Olinto e Marco Antônio Raupp, pelo apoio, compreensão e pela liberação concedida para que eu pudesse continuar com tranquilidade o meu trabalho de doutorado;

ao apoio da equipe de suporte do Departamento de Computação Eletrônica, Norma, Wagner, Márcia e Luis Pereira, em particular ao Sergio Ricardo cuja presteza e atenção com relação a manutenção do IBM/SP permitiram executar os meus testes mesmo nas situações mais difícies;

ao Sergio Guedes, que durante o pouco tempo em que esteve ajudando o grupo de suporte do SP/IBM do LNCC trouxe soluções fundamentais para a implementação do sistema DSMIO no SP;

aos professores da minha linha de pesquisa Edil, Valmir, Victor e Inês pelo conhecimento que trocamos e pela convivência proveitosa de tantos anos;

em especial, à Marta, pela paciência e atenção em me passar o conhecimento do sistema GOA, pelas sugestões ao longo do desenvolvimento da minha tese, pelo estímulo e apoio incondicional nas horas em que mais precisei, mas principalmente pelo exemplo, enquanto pesquisadora;

ao Ricardo, em especial, pelo tema da minha tese, pelas sugestões ao longo do desenvolvimento da mesma, e pelo apoio e incentivo no desenvolvimento do relatório técnico, que possibilitou uma mudança no rumo da minha tese;

à Cláudia, Solange, Sueli, Mercedes, Lúcia e Marli, que fazem parte do corpo administrativo do Programa de Engenharia de Sistemas e Computação, que com dedicação e competência ajudam a construí-lo a cada dia e a mantê-lo. Agradeço também aos técnicos, Júlio(s), Adilson, César, Carlos e Frederico, pela dedicação à manutenção do sistema da PESC;

à Cláudia, Mara e Ana Paula Prata, pela alegria e descontração que tanto nos ajudam a aliviar as tensões do dia a dia;

a todos os participantes das "reuniões de quarta-feira do Ricardo", Luiz Favre, Vinicio, Sílvio, Eduardo, Clicia, Cristiana, Raquel, Lauro e Rodrigo, Inês e Victor. Suas sugestões e comentários tiveram grande influência nos rumos e no desenvolvimento da minha tese e nossas reuniões tiveram contribuição fundamental na minha formação como pesquisadora.

ao Jorge e ao Flávio por terem passado o conhecimento do código fonte do GOA.

aos meus companheiros de sala Cristiana, Clicia, Luiz Favre e Ayru, e mais tarde Raquel, Rodrigo, Lauro, Elcio, Edson e Marcos, pela amizade, pelos desabafos, pelas longas conversas para tentar esquecer momentaneamente nossos intermináveis *bugs*, e pelo incentivo nas horas onde tudo parecia que não ia ter fim;

à Raquel, Clicia e ao Lauro, pelas longas conversas que ajudaram a retirar minhas dúvidas em momentos tão importantes;

à Cristiana, pessoa especial no desenvolvimento da minha tese, por ter me passado todo o seu conhecimento relativo aos fontes do *TreadMarks*, pelas revisões do relatório técnico que possibilitou a mudança no rumo da minha tese, pela consultoria durante o projeto, e por fim pelo companheirismo e amizade;

a toda minha família pelo apoio e principalmente pelo incentivo; pelo incentivo " real " do mano Joe e pelo incentivo " virtual " das manas Ellen e Annick, e do mano Weber;

à Maria, em especial, minhas filhas não poderiam ter tido melhor "mãe" durante a minha ausência.

às duas flores da minha vida, Julia e Sofia, que aguentaram juntas com muita compreensão e bom-humor todas as minhas ausências. Às duas eu agradeço principalmente por me fazerem esquecer, por vários momentos, da existência de *software DSMs* e protocolos de coerência.

a Ignacio, meu marido e companheiro, agradeço por estar ao meu lado em todos os momentos difíceis e decisivos da minha vida. Seu amor e carinho fazem com que isso tudo valha a pena.

e finalmente, um agradecimento muito especial àqueles que foram os meus maiores incentivadores. Àqueles que sempre me apoiaram incondicionalmente e que me deram exemplo de coragem e determinação, a quem eu dedico essa tese, meus pais Aída e Linton.

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

PROPOSTA E AVALIAÇÃO DE MECANISMOS DE *SOFTWARE* DE MEMÓRIA COMPARTILHADA DISTRIBUÍDA PARA E/S PARALELA

Carla Osthoff Ferreira de Barros

Março/2000

Orientador: Claudio Luís de Amorim
Programa: Engenharia de Sistemas e Computação

Aplicações paralelas de diversas áreas, tais como computação científica e banco de dados, necessitam de sistemas de entrada/saída (E/S) com alto desempenho. Esta tese apresenta uma proposta e uma avaliação dos mecanismos adequados nos sistemas de memória compartilhada distribuída baseada em *software (software DSM)* para otimizar o *caching* dos dados de disco e, como conseqüência, melhorar substancialmente o desempenho da E/S em sistemas paralelos. Mais especificamente, a principal contribuição da tese é apresentar mecanismos que permitam: (a) utilizar um modelo de consistência de liberação preguiçoso, LRC, de consistência de memória nos acessos à cache de disco compartilhada, (b) utilizar os mecanismos de *diffings* encontrados em *Software DSM* para reduzir sincronizações, através da existência de múltiplos escritores concorrentes aos dados, economizando espaço de armazenamento da *cache* e diminuindo o tamanho das mensagens de manutenção de coerência do sistema, e (c) permitir unir a facilidade de programação do ambiente de Sistema Gerente de Banco de Dados (SGBD) paralelo de memória compartilhada com o desempenho oferecido pelo ambiente SGBD paralelo de um sistema distribuído. Para avaliar esses mecanismos, desenvolvemos o sistema DSMIO e o testamos em diversas arquiteturas de SGBD paralelos e comparamos o desempenho dos sistemas resultantes utilizando um multicomputador IBM-SP.

Abstract of Thesis presented to COPPE as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

PROPOSAL AND EVALUATION OF DISTRIBUTED SHARED MEMORY CONCEPTS AND MECHANISMS FOR PARALLEL I/O

Carla Osthoff Ferreira de Barros

March/2000

Advisor: Claudio Luís de Amorim

Department: Programa de Engenharia de Sistemas e Computação

Parallel applications from several areas, such as scientific computing and commercial databases, require high-performance input/output (I/O) systems. This thesis proposes and evaluate the exploitation of software-based distributed shared memory (DSM software) concepts and mechanisms to optimize disk caching and, as a result, substantially improve the I/O performance of parallel systems. More specifically, the main contribution of this thesis is a set of mechanisms that enable us: (a) to utilize a lazy release consistency model for disk data accesses; (b) to use diffing mechanisms to save disk cache space, save message coherence size space and eliminate certain types of synchronization, through the existence of multiple concurrent writers to disk data, and (c) to allow a Distributed Object Oriented Parallel Data Base Management System to combine the benefits of shared memory simple programming environment with the performance of distributed memory.

In order to evaluate these mechanisms, we implemented the DSMIO system for a series of parallel database management architectures and compared their performance through an IBM-SP multicomputer system.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Introdução

Sistemas para a movimentação de dados entre a memória primária e os dispositivos externos, chamados comumente de sistemas de entrada e saída (E/S), são parte importante de qualquer plataforma computacional. De fato, alto desempenho do sistema de E/S é crucial para aplicações de diversas áreas, tais como computação científica e bancos de dados. No entanto, a pesquisa em sistemas de E/S mostra que tem sido difícil atingir esse alto desempenho, especialmente para aplicações paralelas.

Em vista disso, um grande número de pesquisas vem sendo realizadas na tentativa de reduzir a latência e aumentar o *throughput* do sistema de E/S [66, 68]. Apesar de todos os avanços nessa área, o tempo médio de acesso aos dados em disco, por exemplo, ainda domina a execução de uma grande classe de aplicações paralelas. Para tal classe, novas otimizações poderiam vir de uma melhor utilização da memória principal para *caching* dos dados de disco, já que o tempo médio de acesso à memória primária é várias ordens de grandeza menor do que o tempo médio de acesso à memória secundária (disco).

Assim sendo, nesta tese propomos explorar os mecanismos encontrados nos sistemas de memória compartilhada distribuída *Distributed Shared Memory, DSM* baseada em *software* para otimizar o *caching* dos dados de disco, de forma a aumentar substancialmente o desempenho da E/S em sistemas paralelos.

Nossa proposta vem da observação de que sistemas DSM (ver figura 1.1) e sistemas de arquivos paralelos e/ou distribuídos (ver figura 1.2) têm formas semelhantes de abstrair a localização dos dados acessados. Mais especificamente, sistemas DSM provêem a abstração de uma memória compartilhada em um ambiente distribuído,

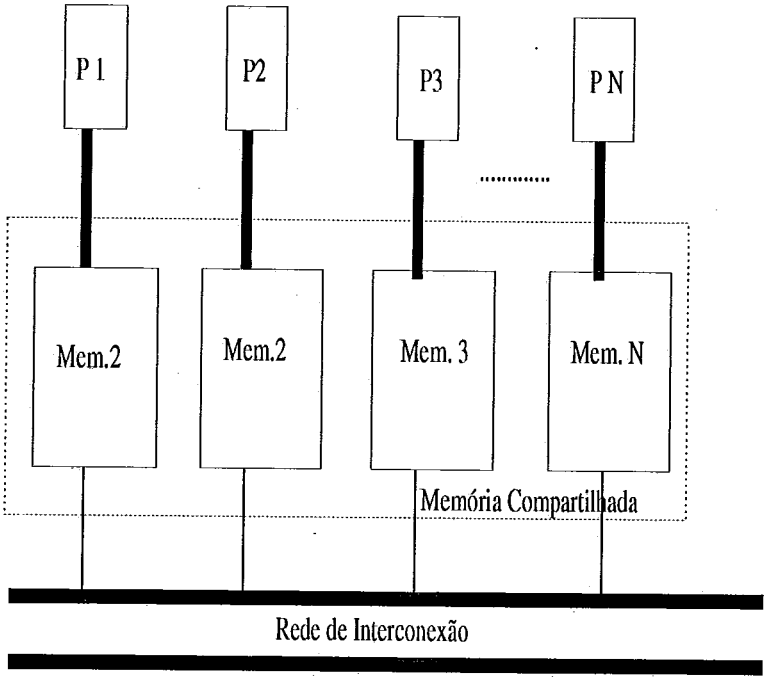

Figura 1.1: Modelo de memória de um sistema DSM

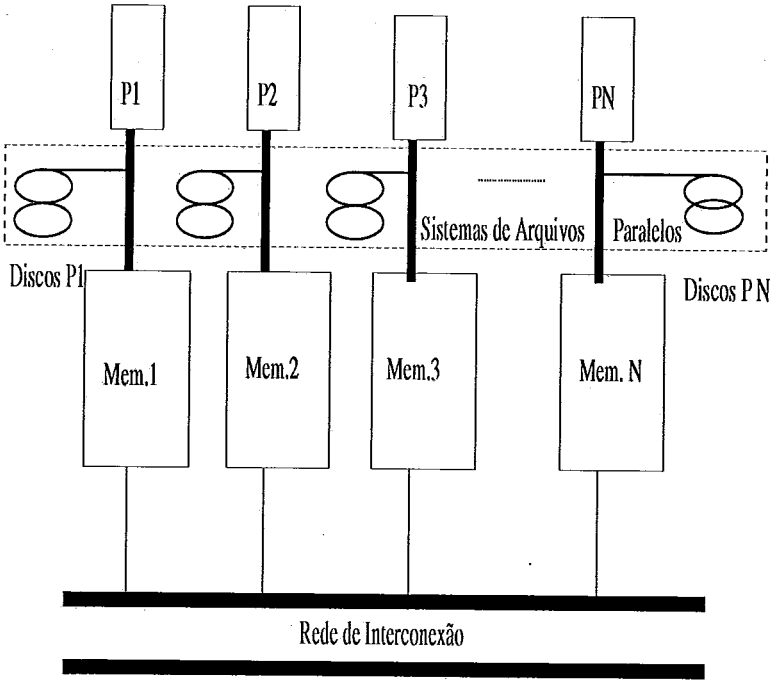

Figura 1.2: Modelo de acesso aos discos de um sistema de Arquivos Paralelos

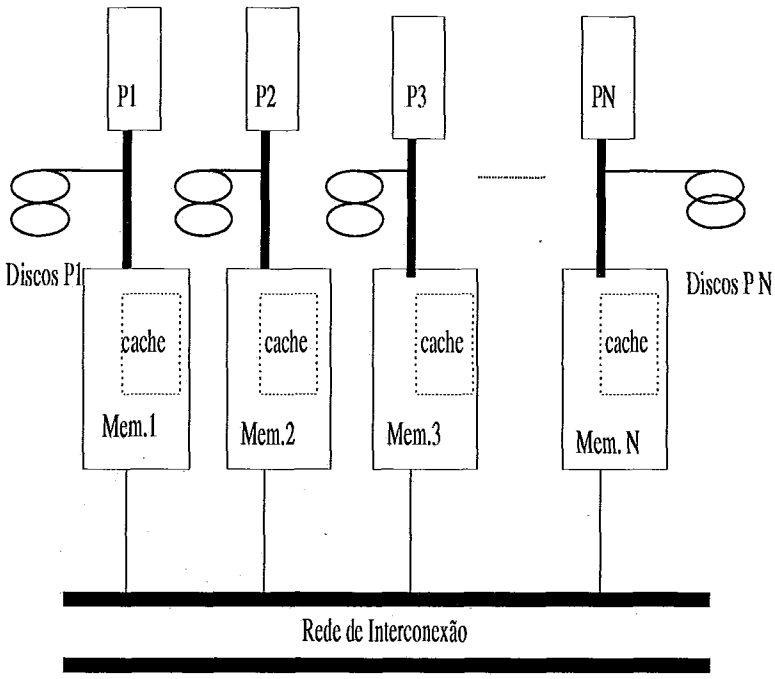

Figura 1.3: Modelo de memória *cache* de E/S de um sistema tradicional

onde as memórias distribuídas são mapeadas num espaço de endereçamento único de forma transparente à aplicação. Um sistema de arquivos paralelo e/ou distribuído fornece uma abstração análoga no nível da memória secundária, uma vez que dados armazenados em qualquer disco podem ser acessados transparentemente pela aplicação. Em um sistema de *cache* de disco tradicional (ver figura 1.3) cada servidor de arquivo possui uma *cache* independente para armazenar os dados do disco. Neste esquema, quando há compartilhamento, a coerência dos dados armazenados nas *caches* locais de cada processador só é garantida no próprio disco [77] e o desempenho de aplicações com altas taxas de escrita de dado se torna muito pequeno [78].

Nesta tese, mostramos que podemos minimizar esse tipo de problema estendendo os mecanismos utilizados em sistemas DSM para garantir a coerência dos dados no nível da memória principal. O foco principal dessa extensão é que todas as *caches* de disco do sistema devem formar abstratamente uma *cache* única, compartilhada por todos os processadores (ver figura 1.4). Além disso, estamos utilizando um modelo relaxado (preguiçoso) de consistência de memória no acesso a essa *cache* compartilhada, o que faz com que ocorra uma redução na comunicação necessária para manter essas memórias consistentes. Permitimos ainda múltiplos escritores concorrentes aos blocos de disco através do mecanismo de *diffing*, onde apenas as atualizações feitas a um dado pertencente ao bloco são mantidas localmente e não o dado inteiro.

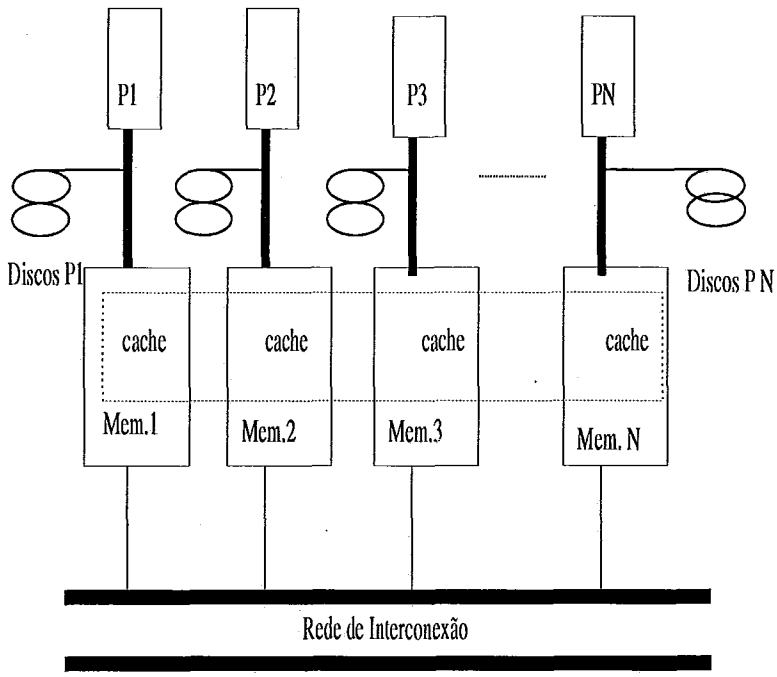

Figura 1.4: Modelo de memória do Sistema DSMIO

Essa última otimização permite que economizemos espaço de armazenamento em memória principal.

Para avaliar essas idéias, construímos o sistema *Distributed Shared Memory I/O*, DSMIO para um multicomputador IBM-SP. A idéia básica do sistema DSMIO é integrar o sistema de *cache* de disco a um sistema *software DSM*, no caso o sistema TreadMarks [52]. A integração entre esses dois sistemas, na verdade, pode ser vista por dois ângulos distintos: o sistema DSMIO provê ao sistema TreadMarks a possibilidade de realizar E/S de forma otimizada; ou TreadMarks provê ao sistema DSMIO uma memória compartilhada e mecanismos de coerência que permitem a implementação de uma *cache* compartilhada.

Além do sistema DSMIO utilizamos aplicações alvo que necessitam de um acesso otimizado de E/S. Em particular, aplicações onde ocorra compartilhamento de dados e atualização nos dados compartilhados, como no caso da classe de aplicações de Sistemas Gerenciadores de Banco de Dados (SGBDs).

As aplicações de SGBDs têm evoluído de tal forma que geraram a necessidade de se analisar dados das mais variadas fontes, através de consultas complexas efetuadas sobre um grande volume de dados em tempo hábil. Assim, para se obter um melhor desempenho dos SGBDs, pesquisas vem sendo desenvolvidas nas áreas de sistemas gerenciadores de bancos de dados distribuídos e paralelos. Concomitantemente,

novos domínios de aplicação surgiram, exigindo modelagem e funcionalidades não atendidas satisfatoriamente pelos SGBDs relacionais. Essas novas classes de aplicações ditas não convencionais, tais como *Computer Aided Design* (CAD), *Computer Aided Sofwtare Engineering* (CASE) e Sistemas de Informações Geográficas (SIG), fomentaram o desenvolvimento de novas tecnologias de bancos de dados, em especial os Sistemas Gerenciadores de Bancos de Dados Orientados a Objetos, SGBDOOs [14] e os Relacionais Objetos(SGBDROs).

Devido à ampla disseminação e representatividade da classe de aplicações relacionadas com os sistemas SGBDOOs, avaliamos o desempenho do nosso sistema como uma extensão de um sistema de gerência de banco de dados (SGBD) orientado a objetos. Utilizamos o SGBDOO desenvolvido pela COPPE/UFRJ, denominado GOA [15].

Para avaliar o desempenho do sistema DSMIO desenvolvemos quatro testes com o objetivo de comparar o sistema em relação a quatro plataformas distintas de sistemas SGBD distribuídos, que estão discriminados a seguir:

- Sistema DSMIO *versus* Plataforma Tradicional.

  Inicialmente comparamos o desempenho do sistema SGBD DSMIO em relação a um ambiente com SGBD paralelo tradicional, onde os processadores acessam um base localizada em um mesmo disco, via sistema NFS através de uma rede *ethernet*. O objetivo deste teste está em avaliar a eficiência de se trazer a coerência da *cache* do nível do disco para o nível da memória primária. Nossos resultados demonstram que ao trazer a coerência dos dados armazenados em disco ao nível da memória principal empregando os mecanismos do *software DSM*, aumenta significativamente o desempenho do sistema de entrada/saída.

- Sistema DSMIO *versus* Plataforma SGBD que implementa um modelo de consistência de *cache Release Consistency*.

  Este teste compara o sistema SGBD DSMIO em relação a um ambiente SGBD que mantêm a consistência de *cache* segundo um modelo menos relaxado que o modelo adotado pelo sistema DSMIO, no caso o modelo *Release Consistency*. O objetivo desse estudo está em avaliar a eficiência de se adotar um modelo *Lazy Release Consistency* e de mecanismos de *diffs* para a otimização de E/S paralela. Nossos resultados mostram experimentalmente que o modelo LRC reduz drásticamente o número de mensagens necessárias para se manter a

coerência de *caches* compartilhadas e que o mecanismo de *diffs* permite uma drástica redução no tamanho das mensagens de coerência enviadas pelo sistema de E/S, assim como uma redução significativa no tamanho da área de memória local necessária para se armazenar os dados a serem escritos no disco.

- Sistema DSMIO *versus* Algoritmo que representa o estado da arte de consistência de *cache* Cliente-Servidor.

  Este estudo tem como objetivo avaliar o potencial do sistema DSMIO na área de aplicações de SGBD. Comparamos o desempenho do sistema DSMIO em um ambiente cliente-servidor em relação ao estado da arte dos algoritmos de consistência de *cache* cliente-servidor [35], no caso o algoritmo de consistência de *cache Call-back Locking* -CBL [8]. Nossos resultados mostram que no contexto dos testes realizados, DSMIO apresenta um algoritmo de consistência de cache muito mais eficiente que o algoritmo de consistência de cache mais adotado para os sistemas SGBD Orientado a Objeto (CBL).

- Sistema DSMIO *versus* Algoritmo que representa o estado da arte de consistência de *cache* Cliente-Servidor em um ambiente SGBD Paralelo.

  Comparamos o desempenho do sistema DSMIO em um ambiente cliente-servidor SGBD paralelo em relação ao algoritmo de consistência de *cache* cliente-servidor SGBD paralelo CBL. Os experimentos evidenciam que o sistema DSMIO em um ambiente com SGBD paralelo permite unir a facilidade de programação do ambiente SGBD paralelo de memória compartilhada com o desempenho oferecido pelo ambiente SGBD paralelo de sistemas distribuídos.

1.2 Contribuições da Tese

Sumariamente, as principais contribuições desta tese são:

- Resultados experimentais que demostram que o uso de mecanismos de sistemas *software DSM* melhoram significativamente o desempenho de E/S em aplicações paralelas. Mais especificamente:

  - Proposta e avaliação de utilização de um modelo *Lazy Release Consistency* (LRC) de consistência de memória nos acessos a *cache* compartilhada;

- Proposta e avaliação da utilização do mecanismo de *diffing*, que permite a existência de múltiplos escritores concorrentes e uma possível redução de sincronizações, para aumentar o desempenho do sistema, possibilitando diminuir o tamanho das mensagens assim como o tamanho da área de memória local necessária para armazenar os dados modificados.

- Propor e avaliar um novo algoritmo de consistência de *cache* eficiente em um ambiente SGBD cliente-servidor, e em ambientes SGBD cliente-servidor paralelos.

- Demonstrar experimentalmente que o sistema DSMIO é capaz de fornecer desempenho para E/S em sistemas *software DSM* de modo que tais sistemas venham a se tornar uma opção capaz de fornecer um ambiente de programação eficiente e amigável para uma arquitetura SGBDOO paralela de memória distribuída.

1.3 Organização da Tese

Esta tese está organizada da seguinte forma. No capítulo 2, apresentamos conceitos básicos de sistemas *software DSM*, em particular apresentamos o sistema *TreadMarks*. Descrevemos o sistema DSMIO no capítulo 3. No capítulo 4, revisamos as principais propostas de arquiteturas paralelas de SGBD relacionadas à área de Banco de Dados. No capítulo 5, avaliamos o desempenho do sistema DSMIO em um SGBD isolando os efeitos de mecanismos de *software DSM*. Em seguida, no capítulo 6, propomos um novo algoritmo de consistência de *cache* baseado em DSMIO para um sistema cliente-servidor. No capítulo 7, avaliamos a proposta do sistema DSMIO em um sistema cliente-servidor. No capítulo 8, avaliamos o desempenho do sistema cliente-servidor DSMIO. No capítulo 9, propomos e avaliamos o desenvolvimento de um sistema paralelo cliente-servidor DSMIO. No capítulo 10, discutimos os trabalhos relacionados. Finalmente, no capítulo 11 apresentamos as nossas conclusões e os trabalhos futuros.

Capítulo 2

Sistemas DSM

2.1 Sistemas DSM

Sistemas com memória compartilhada distribuída, ou *DSM (Distributed Shared Memory)*, implementam o modelo de programação de memória compartilhada em ambientes distribuídos como podemos ver na figura 1.1. O grande atrativo destes sistemas está na possibilidade de unir um modelo de programação simples a um *hardware* escalável.

Em sistemas DSM, a memória compartilhada é implementada por mecanismos de *hardware* e/ou *software* que transformam, de modo transparente ao usuário, os acessos às memórias remotas em mensagens pela rede. Sistemas *hardware DSM* implementam a memória compartilhada através de mecanismos de *hardware* altamente especializados que permitem grande eficiência, mas tornam o sistema complexo e extremamente caro [62]. Sistemas *software DSM* são uma alternativa de baixo custo. Esses sistemas normalmente implementam a memória compartilhada através do mecanismo de proteção de páginas do *hardware* de memória virtual presente em qualquer arquitetura convencional. Entretanto, tais sistemas apresentam desempenho limitado devido principalmente ao grande *overhead* gerado pelo protocolo de coerência empregado. Sistemas de *Software DSMs* baseados em modelos de consistência relaxada tentam reduzir esse *overhead* atrasando e/ou restringindo ao máximo a comunicação e as transações de coerência, como no caso do sistema de *software DSM TreadMarks* [52]. *Treadmarks* é o sistema *software DSM* mais utilizado nos dias de hoje e é o sistema base utilizado pelo sistema DSMIO.

Iniciamos este capítulo revendo alguns conceitos básicos de *Software DSM*. Discutimos os modelos de consistência, e a seguir faremos uma descrição do protocolo de coerência implementado pelo sistema de *Software DSM TreadMarks* [52].

2.2 Modelos de Consistência

Os modelos de consistência de memória de um sistema multiprocessador de memória compartilhada especificam como os acessos à memória são observados pelos processadores.

Essencialmente, os modelos de memória restringem os valores que uma operação de leitura pode retornar. Intuitivamente, a leitura deve retornar o valor da última operação de escrita executada na posição de memória sendo lida. Em um sistema uniprocessador o conceito de "último" é definido pela ordem seqüencial especificada pelo programa, chamada de "ordem do programa". O modelo de consistência seqüencial é uma extensão intuitiva deste modelo. Ele requer que todas as operações de memória apareçam para o programador como executando uma por vez e que cada uma das operações sendo executadas em um cada um dos processadores apareça como executando na ordem descrita pelo programa do processador. O modelo de consistência seqüencial fornece um modelo de programação simples e intuitivo, entretanto ele impede a implementação de algumas otimizações consideradas importantes nos processadores atuais. A utilização de *write-buffers*, *pipelines* de escrita, e *overlap* de operações de memória estão entre as otimizações comuns que podem violar o modelo[39]. Vários modelos de consistência de memória relaxados foram propostos para viabilizar o uso de algumas otimizações no acesso à memória compartilhada, e assim, com um modelo menos restritivo, melhorar o desempenho do sistema. Nas seções seguintes apresentamos a descrição de alguns modelos de consistência de memória. Nossa preocupação aqui não é a de apresentar os formalismos da definição de cada modelo, mas sim dar uma noção intuitiva do funcionamento de cada modelo num sistema DSM. Definições formais podem ser encontradas em [38] [39] [25].

2.2.1 Modelo de Consistência Seqüencial

O modelo de consistência seqüencial foi introduzido por Lamport [45], através de condições para o correto funcionamento de uma máquina com memória compartilhada. Um multiprocessador ou, genericamente, um sistema de memória compartilhada é definido como seqüencialmente consistente se o resultado de qualquer execução é o mesmo resultado obtido caso qualquer das operações de todos os processos fossem executadas em qualquer ordem seqüencial. Além disso, as operações de cada pro-

cessador aparecem nesta seqüencia na mesma ordem que foram especificadas pelo respectivo programa paralelo.

Lamport definiu duas regras para que uma máquina multiprocessada seja seqüencialmente consistente.

- Cada processador envia pedidos de acesso a memória na ordem especificada pelo seu programa.
- Pedidos de acessos emitidos para um mesmo módulo de memória têm que ser atendidos numa ordem FIFO.

Uma escrita é executada com relação a um processador Pi quando leituras feitas por Pi retornam o novo valor escrito. Leituras são executadas com relação a um processador Pi quando uma escrita feita por Pi não pode mais afetar o valor retornado pela leitura. Quando um acesso de leitura ou escrita é executado com relação a todos os processadores, então dizemos que o acesso foi globalmente executado. Além disso, uma leitura é globalmente executada quando esta foi executada e o acesso de escrita que gerou o valor retornado foi também executado.

As regras acima se referem a características de *hardware*. Entretanto, pode-se definir a seguinte condição , relativa a ordenação de eventos, necessária para garantir a consistência seqüencial [25].

- Antes que uma leitura ou escrita possa ser executada com relação a qualquer outro processador, todos os acessos de leitura anteriores têm que ter sido globalmente executados e todos os acessos de escrita anteriores têm que ter sido executados.

2.2.2 Modelo de Consistência por Processador

O modelo *Processor Consistency* (PC) foi introduzido por Goodman[43] e elimina algumas das restrições do modelo de consistência seqüencial na medida em que exige apenas que as operações de cada processador sejam executadas na mesma ordem especificada pelo programa. Isto significa que a ordem em que escritas de dois processadores distintos são observadas por quaisquer processadores não necessitam ser idênticas. O argumento básico para a utilização deste modelo é que a sua implementação não é tão cara quanto a consistência seqüencial e, para a maioria das

aplicações, um programa que espera um modelo seqüencial apresenta os mesmos resultados no modelo por processador . Isto porque este modelo exige a implementação de sincronizações explícitas para garantir a correta execução do programa.

Este modelo garante que as escritas sejam observadas de modo consistente em cada processador e permite que algumas operações de leitura ultrapassem os acessos de escrita (desde que para posições de memória diferentes destas). Com isso, existe a oportunidade de utilização de *write buffers* e *pipelining*, o que permite a obtenção de um desempenho melhor que no modelo de consistência seqüencial.

2.2.3 Modelo de Consistência Fraca

Este modelo foi denominado *Weak Consistency* (WC) e proposto por Dubois, Scheurich e Briggs [41].

A mesma idéia utilizada no modelo de consistência por processador de se considerar as sincronizações explícitas presentes nos códigos de programas concorrentes é também aplicada no modelo de consistência fraca. No modelo de consistência seqüencial, se vários acessos de escrita são requisitados por determinada computação, então cada acesso tem que ser atrasado até que todos os acessos anteriores tenham sido completamente executados. Entretando, estes atrasos são desnecessários porque, em geral, o programador pode proteger os acessos conflitantes através de mecanismos de sincronização. Uma vez que todos os pontos de sincronização são identificados, basta que a memória compartilhada esteja consistente nestes pontos para garantir o correto funcionamento dos programas.

Ele é o primeiro modelo relaxado de consistência considerando a relação entre a ordem dos pedidos de acesso à memória e os pontos de sincronização do programa. Nele, as operações de sincronização funcionam como cercas (*fences*) que obedecem ao modelo de consistência seqüencial. Esse modelo estabelece que acessos a dados compartilhados realizados entre dois acessos de sincronização podem ser observados por outros processadores em qualquer ordem. Quando um acesso de sincronização é realizado, os acessos anteriores devem ter sido observados por todos os processadores.

A vantagem deste modelo é permitir que acessos a memórias com grandes latências possam ser sobrepostos com outros acessos, permitindo um melhor desempenho dos programas. A desvantagem é que todos os acessos relativos a sincronização devem ser identificados pelo programador ou pelo compilador. As condições para se assegurar a consistência fraca são as seguintes:

- Antes que um acesso comum possa ser executado com relação a qualquer outro processador, todos os acessos de sincronização têm que ter sido executados;
- Antes que um acesso de sincronização possa ser executado com relação a qualquer outro processador, todos os acessos comuns anteriores têm que ter sido executados;
- Acessos de sincronização são seqüencialmente consistentes entre si.

Por permitir uma maior possibilidade de reordenação dos acessos a memória, os sistemas que implementam o modelo de consistência fraca têm um potencial de ganho de desempenho maior do que aqueles sistemas que implementam os modelos de consistência seqüencial e consistência por processador.

2.2.4 Modelo de Consistência por Liberação

O modelo de Consistência por Liberação (*Release Consistency* /RC), foi introduzido por Gharachorloo [42] e assim como outros modelos de consistência fraca, procura fazer com que acessos que produzam alterações na memória compartilhada tenham seus efeitos retardados. Para tal, os acessos são rotulados como ordinários ou especiais. Dentre os especiais, os acessos são subdivididos naqueles que são de sincronização e naqueles que não. É responsabilidade do compilador ou do programador classificar os acessos baseados nas propriedades intrínsecas de cada acesso. A idéia básica é que se os pontos de sincronização podem ser identificados, então somente nestes pontos a memória precisa estar consistente. O modelo de consistência por liberação requer apenas que as operações tenham terminado antes que uma liberação de uma região crítica seja efetuada.

Um sistema é consistente por liberação se:

- Antes que um acesso tenha permissão de execução com relação a qualquer outro processador, todas as aquisições anteriores têm que ter sido efetuadas;
- Antes que uma liberação tenha permissão de execução com relação a qualquer outro processador, todas as leituras e escritas ordinárias têm que ter sido efetuadas;
- Acessos especiais são seqüencialmente consistentes entre si.

Programas com acessos adequadamente sincronizados, ou seja, programas cujos dados compartilhados estão devidamente protegidos dentro de regiões críticas por variáveis de sincronização, produzem os mesmos resultados sobre o modelo de consistência por liberação que produziriam sobre o modelo de consistência seqüencial [42]. Informalmente, um programa adequadamente sincronizado possui acessos com aquisições ou liberações, ou seja, pontos de sincronização, em número suficiente de forma que, para todos os entrelaçamentos legais de acessos, cada par de acessos ordinários conflitantes é separado por uma cadeia de liberação/aquisição. Dois acessos conflitam entre si se eles referenciam a mesma posição de memória e pelo menos um deles é de escrita.

A definicão de um programa propriamente sincronizado é a seguinte:

- Sejam **u** e **v** dois acessos a um mesmo dado compartilhado, **u** é realizado no processador **Pu** e **v** no processador **Pv**. Para que em qualquer execução do programa, **v** seja visto pelo outros processadores antes de bf u, deve haver em **Pu** uma leitura de uma variável de sincronização e em **Pv** uma escrita na mesma variável, tal que **Pu** leia o valor escrito em **Pv**. Um programa é propriamente sincronizado se essa condição vale para todos os possíveis pares **u** e **v**.

De uma forma simplificada, uma aplicação é propriamente sincronizada se ocorrem sincronizações suficientes para evitar condições de corrida, ou seja, condic cões em que o dado pode ficar em um estado inconsistente devido ao compartilhamento por operações de escrita entre os processadores que compartilham o dado.

A constatação de que para programas devidamente sincronizados, o modelo de consistência por liberação é equivalente ao modelo de consistência seqüencial é um importante resultado uma vez que os programadores podem utilizar a semântica do modelo de consistência seqüencial utilizando na verdade um modelo que não tem tanto impacto no desempenho das aplicações.

A subdivisão de acessos de sincronização em aquisição e liberação permite ao modelo de consistência relaxada um tratamento diferenciado e mais flexível do que o empregado para as cercas do modelo de consistência fraca.

O modelo de consistência por liberação, diminui as restrições de consistência com relação ao modelo de consistência fraca, e ainda executa corretamente programas propriamente sincronizados, necessitando porém que os acessos de sincronização

sejam mapeados em aquisições e liberações. Este fato faz com que a programação tenha que ser um pouco mais cuidadosa, porém, apresenta uma redução sensível na latência de acesso à memória compartilhada, pois os pontos de espera são apenas nas operações de liberação. Como exemplos de sistemas que utilizam o modelo de consistência relaxada podemos citar DASH[46], Quarks[53] e Shasta[49].

2.2.5 Modelo de Consistência por Liberação Agressivo

Na implementação do modelo de consistência por liberação de Munin[40], um processador atrasa as suas modificações em dados compartilhados até o momento da liberação da região crítica. Neste instante, ele difunde as modificações para todos os demais processadores que armazenam o objeto modificado. Se um protocolo de invalidação é utilizado, isto é efetuado simplesmente através do envio de invalidações para todos os objetos replicados. De qualquer modo, a liberação é bloqueada até que todos os *acknowledgements* sejam recebidos de todos os processadores do sistema.

2.2.6 Modelo de Consistência por Liberação Preguiçoso

Nesta variação, o modelo de consistência por liberação é alterado para que as modificações em dados compartilhados sejam difundidas no momento da execução de uma operação de aquisição de uma variável de sincronização, por exemplo um *lock*, de uma região crítica e não no momento da operação de liberação do *lock*. Neste instante, o processador requisitante determina quais modificações ele necessita receber de acordo com a definição da região crítica. Desta forma, um processador que adquira um *lock* irá receber todas as modificações que precedem a aquisição daquele *lock*. Uma definição formal pode ser encontrada em [1].

O modelo *Lazy Release Consistency* (LRC)[42] é uma versão mais relaxada do modelo de consistência por liberação, suportando o mesmo modelo de programação. No modelo LRC, as condições de consistência não necessitam que numa operação de liberação os acessos anteriores sejam globalmente visíveis. Ele necessita apenas que os acessos anteriores à operação de liberação sejam visíveis somente no processador que executar um operação de aquisição. Isto é, a propagação das modificações feitas por um processador P_i para outro processador P_j são atrasadas até o momento de uma aquisição subseqüente em P_j numa mesma variável de sincronização.

Desta forma, não há uma operação de difusão das modificações no momento da operação de liberação. A comunicação é realizada somente entre os dois processa-

dores envolvidos nas operações de aquisição e liberação. Este fato faz com que a operação de aquisição tenha um potencial reduzido de comunicação.

Para determinar quais modificações um processador deve ter ciência no momento de uma aquisição, o modelo LRC estabelece uma ordem parcial dos acessos aos dados compartilhados, denominada *happened-before-1* (*hb*1). Esta ordem parcial *hb*1 é baseada na ordem seqüencial de execução de um processador e no encadeamento das operações de aquisição e de liberação de um *lock* realizadas em processadores diferentes, mas sob a mesma variável de sincronização. Dois acessos à memória compartilhada a_1 e a_2 são ordenados por *hb*1, denotado por $a_1 \overset{hb1}{\rightarrow} a_2$, se:

- a_1 e a_2 são acessos do mesmo processador e a_1 ocorre antes de a_2;
- a_1 é um *release* no processador P_1, a_2 é um *acquire* na mesma variável de sincronização em P_2 e a_2 retorna o valor escrito por a_1.

A ordem *hb*1 é dada basicamente pelo fecho transitivo da ordem de execução de um processador com a ordem das operações de sincronização na mesma variável de sincronização. Se $a_1 \overset{hb1}{\rightarrow} a_2$ e $a_2 \overset{hb1}{\rightarrow} a_3$ então $a_1 \overset{hb1}{\rightarrow} a_3$.

Como exemplo,entre os sistemas DSM que implementam o modelo de consistência LRC podemos citar TreadMarks[52], AURC[44] e ADSM[48].

O Protocolo Home Based Lazy Release Consistency

Uma outra forma preguiçosa,(*lazy*), de propagação de modificações é realizada pelo protocolo *Home Based Lazy Release Consistency* (HLRC) [69]. Este protocolo implementa o modelo LRC de forma diferente do apresentado acima. Neste protocolo a propagação é executada em dois estágios e um nó *home* é selecionado para cada página compartilhada, de forma que no primeiro estágio as modificações, os *diffs*. *Diff* é uma estrutura contendo a diferença entre uma página original e uma página modificada, explicado em detalhes na seção 2.3.4, são propagadas para os nós *home* de forma agressiva, ou seja, durante ou antes da liberação do bloqueio. Os nós *home*, ao receberem os *diffs* dos seus clientes, aplicam imediatamente às páginas relacionadas, para então serem eliminados.

Os nós que não são *homes* mantêm a coerência das suas cópias através de um protocolo de invalidações, ou seja, em um segundo estágio, quando ocorre uma falha de acesso a uma página o processador solicita a página completa ao seu respectivo nó

home, ao invés de solicitar os *diffs* aos escritores anteriores. As vantagens principais com relação ao desempenho são:

- Menos mensagens, já que as falhas de acessos podem ser satisfeitas com apenas um pedido/recebimento de uma mensagem para o nó *home*.
- O *diff* é aplicado apenas uma vez (no seu respectivo nó *home*).
- As escritas executadas no nó *home* não geram *diffs*.
- Não existe busca remota de dados ao nó *home*.

As desvantagens são:

- A busca de uma página inteira quando apenas uma palavra é modificada.
- Baixo desempenho caso as cargas nos nós *home* não sejam devidamente balanceadas.

Modelo de Consistência por Entrada

O modelo de consistência por entrada(*Entry Consistency*/ EC), foi desenvolvido por Bershad *et al.*[55] dentro do projeto do sistema *Midway*. Ele relaxa a consistência dos dados em relação a LRC através da associação dos dados compartilhados a variáveis de sincronização. Assim como no LRC, no modelo EC todos os dados compartilhados têm que ser explicitamente rotulados e as ações de consistência são tomadas no momento em que uma aquisição acontece. A diferença básica entre as duas abordagens está no fato de que o modelo EC exige ainda que todos os dados compartilhados sejam associados à uma variável de sincronização. No modelo EC, somente os dados associados à variável de sincronização são atualizados no momento da aquisição, em contraste com o modelo LRC que atualiza todos os dados compartilhados.

2.3 O Sistema TreadMarks

O sistema *TreadMarks* [52] é um sistema de *software DSM* que adota o modelo de consistência LRC, objetivando uma maior redução de comunicação para se manter a coerência dos dados compartilhados. Ele emprega a página como unidade de

coerência, utiliza protocolo de invalidação para a manutenção da coerência dos dados compartilhados e provê suporte a múltiplos escritores através de mecanismo de *diffs*. Desta forma, dois ou mais processadores podem escrever na sua cópia local de uma mesma página, desde que em regiões distintas da página, e as modificações feitas por todos os processadores serão unidas na próxima operação de sincronização, de acordo com a definição do modelo LRC.

A seguir vamos apresentar mais detalhes sobre características do protocolo do sistema *Treadmarks*.

2.3.1 Unidade de Coerência

Sistemas *software DSM* normalmente utilizam uma página ou um objeto como unidade de coerência. No entanto, alguns sistemas *software DSM*, como Shasta [49] e Blizzard-S [57], permitem o emprego de uma granularidade mais fina como unidade de coerência, mas necessitam o uso de sistemas de comunicação de altíssimo desempenho.

No caso dos sistemas que adotam a página como unidade de coerência, como no caso de *Treadmarks*, a sua implementação é fornecida pelo *hardware* de memória virtual, presente em qualquer arquitetura convencional. Uma página protegida contra leitura está com conteúdo inválido e o primeiro acesso a essa página causa uma `falha de acesso`. O tratamento dessa falha é realizado pelo sistema *software DSM*, que se encarrega de produzir uma versão atualizada da página. A detecção de escritas realizadas em uma página é feita de forma simples através do mecanismo de proteção de páginas. As páginas são inicialmente protegidas contra escrita. Na primeira falha de escrita, o sistema identifica que a página será alterada, guarda essa informação e desprotege a página para escritas posteriores.

2.3.2 Protocolo de Invalidação

Existem duas formas diferentes de se notificar um processador sobre modificações realizadas no dado compartilhado: por protocolo de invalidação ou por protocolo de atualização.

Num protocolo de invalidação, um processador é informado a respeito das modificações em um dado compartilhado através de mensagem de invalidação. Ao receber a mensagem o sistema invalida a unidade de coerência que contém o dado. A transferência das modificações realizadas no dado se dá efetivamente no próximo

acesso realizado ao dado. Num protocolo de atualização um processador informa a outro processador a respeito de modificações no dado compartilhado enviando-lhe o próprio dado modificado. O uso de protocolo de atualização evita a ocorrência de falhas de acesso mas aumenta bastante a quantidade de tráfego desnecessário na rede (muitas vezes o processador recebe modificações que não vai utilizar [56]), em relação ao protocolo de invalidações, mas pode ser melhor para algumas aplicações.

2.3.3 Intervalos

No protocolo empregado por *Treadmarks* para executar o modelo LRC, o sistema divide a execução do programa em intervalos para determinar quais modificações no dado compartilhado um processador deve observar no momento da execução de uma aquisição de uma variável de bloqueio.

Intervalos são segmentos de tempo na execução de um processador. Um novo intervalo é iniciado cada vez que uma operação de sincronização é executada. *TreadMarks* utiliza a ordem parcial $hb1$ para ordenar os intervalos de diferentes processadores. Através de $hb1$ é possível determinar quais intervalos de outros processadores precedem o intervalo corrente de um processador p. As modificações realizadas nos dados compartilhados são associadas aos intervalos em que elas ocorreram e, assim, numa operação de aquisição de bloqueio ocorrida no intervalo i, o processador deve ser notificado sobre todas as modificações associadas a intervalos " anteriores " a i segundo a ordem parcial $hb1$.

A notificação sobre as modificações ocorridas nos dados compartilhados é realizada através de *write-notices*, ou avisos de escrita. Um aviso de escrita indica que uma determinada página foi modificada. Cada intervalo contém um aviso de escrita para cada página modificada no segmento de tempo correspondente. Quando o processador p executa uma operação de aquisição de um bloqueio num intervalo i ele deve receber os avisos de escrita correspondentes a todos os intervalos anteriores a i segundo $hb1$. Ao receber um aviso de escrita, o processador invalida a página correspondente. Os *diffs* relativos às modificações em questão só são recebidos na próxima falha de acesso a cada página.

2.3.4 Mecanismos de *diffs*

Considerando a página como unidade de coerência, o falso compartilhamento ocorre quando processadores diferentes escrevem em dados logicamente não relacionados,

mas que estão localizados numa mesma página. Se o acesso de escrita à página for exclusivo, então pode ocorrer o efeito "ping-pong" de uma página quando processadores requisitam alternadamente o acesso exclusivo a ela.

Para evitar o efeito " ping-pong " de uma página, alguns sistemas *software DSM* permitem que vários processadores estejam escrevendo ao mesmo tempo na mesma página, desde que em dados diferentes. No caso de *TreadMarks* isto é permitido através da implementação do mecanismo de criação de *diffs*,definido aqui também.

Antes de começar a atualizar uma página, o sistema cria uma cópia gêmea da mesma, chamada *twin*. As escritas são realizadas livremente na página e o *twin* mantém a versão original. Quando as escritas locais têm que ser propagadas, o processador compara a cópia corrente da página com o seu *twin*, gerando uma estrutura chamada *diff*, as diferenças entre elas. A combinação dos *diffs* gerados por todos os processadores que atualizaram a página permite a formação de uma versão atualizada da mesma.

No *Treadmarks*, a criação de *diffs* é atrasada até o momento em que o processador requisita as modificações feitas na página em questão ou quando um aviso de escrita é recebido para esta página. Neste caso, é importante observar a ordem de computação dos *diffs*.

No momento de uma falha de acesso a uma página devido a página inválida, um processador deve aplicar todos os *diffs* criados durante os intervalos anteriores segundo a ordem parcial. Se um processador P modificou uma página durante o intervalo I, então P tem todos os *diffs* de todos os intervalos anteriores (inclusive os *diffs* relativos a outros processadores). É necessário, então, verificar o maior intervalo de cada processador para qual existe um aviso de escrita mas não exista o *diff* correspondente.

O processador determina então quais os *diffs* que devem ser requisitados e a quais processadores o pedido deve ser feito. Quando todos os *diffs* necessários são recebidos estes são aplicados sobre as páginas correspondentes em ordem crescente de acordo com o valor do seu intervalo.

Para minimizar o número de mensagens necessárias para a busca de *diffs*, *TreadMarks* utiliza esquema de "dominância" de intervalos. Um intervalo i domina um intervalo j, se j precede i na ordem parcial $hb1$. Assim, as mensagens devem ser enviadas apenas aos processadores cujos intervalos mais recentes não são dominados pelos intervalos mais recentes de outros processadores.

2.3.5 Operações de Sincronização

A interface de programação oferecida por *TreadMarks* provê dois tipos de primitivas de sincronização: as primitivas de aquisição e de liberação de uma variável de bloqueio, que são implementadas como *Tmk-lock-acquire/Tmk-lock-release* e as barreiras, implementadas como *Tmk-barrier*. As primitivas *Tmk-lock/Tmk-unlock* são utilizadas para delimitar seções críticas e a primitiva *Tmk-barrier* é utilizada para sincronização global.

Há um processador gerente, determinado estaticamente, para cada variável de sincronização. Numa operação bloqueio em uma variável de sincronização s, o processador p envia uma mensagem ao processador gerente de s (p não conhece a identificação do último processador a acessar s). O processador gerente, então, avança a mensagem para o último nó que executou a operação de liberação do *lock* de s. Este compara seus intervalos com os de p e envia para p os avisos de escrita relativos aos intervalos que p ainda não recebeu informação. Na chegada da mensagem com os avisos de escrita, p invalida as páginas correspondentes.

Numa operação de barreira, cada processador envia para o processador gerente da barreira uma mensagem contendo todos os seus intervalos. O processador gerente " incorpora " todos os intervalos recebidos. Depois que todos os processadores chegaram à barreira, o gerente envia uma mensagem de volta para cada processador p contendo os avisos de escrita relativos aos intervalos dos outros processadores que p ainda não viu. Mais uma vez, na chegada da mensagem com os avisos de escrita, o processador invalida as páginas correspondentes.

2.3.6 Falhas de Acesso

O sistema *TreadMarks* utiliza a função de *mprotect* do sistema operacional *UNIX* para controlar o acesso às páginas compartilhadas. Qualquer violação gera um sinal de interrupção no Sistema Operacional(SIGSEGV).

Inicialmente, todas as páginas do sistema estão válidas no processador 0 e inválidas nos outros processadores. Na primeira falha de acesso a uma página k o processador p ($p \neq 0$) deve requisitar uma cópia de k ao processador 0.

Uma falha de acesso a uma página válida é tratada de modo bem simples. Se a falha ocorreu por causa de um acesso de escrita a uma página que ainda não possui uma cópia da página, ou *twin*, *TreadMarks* deve simplesmente criar o *twin*

correspondente e desproteger a página contra escrita.

Se a falha ocorreu em uma página k inválida, então o sistema tem que trazer os *diffs* relativos às modificações realizadas em k por outros processadores. Os *diffs* devem ser buscados dos processadores que enviaram *write-notices* para k.

Capítulo 3

O Sistema DSMIO

3.1 O Sistema DSMIO

Nesse capítulo descrevemos o sistema DSMIO que permitirá demonstrar como os mecanismos utilizados em sistemas DSM podem ser estendidos para garantir a coêrencia dos dados no nível da *cache* do disco.

3.1.1 Estrutura Geral

O sistema DSMIO utiliza uma área da memória compartilhada estabelecida por *TreadMarks*, chamada de *cache* compartilhada, doravante denominada simplesmente *cache*, para armazenar os dados lidos do disco. Entretanto, as modificações realizadas nos dados são armazenadas na memória local do nó na forma de *diffs*.

A coerência das páginas da cache compartilhada é mantida segundo o modelo LRC de consistência de memória adotado por *TreadMarks*. Assim, o sistema DSMIO mantém toda a estrutura de intervalos e *write-notices* utilizadas por *TreadMarks*. Mais especificamente, o sistema DSMIO atrasa a propagação das informações de coerência relativas às modificações realizadas nos dados da cache até o momento de uma operação *acquire* (entrada de uma seção crítica), quando o processador recebe os *write-notices* relativos às páginas escritas e invalida suas respectivas páginas. Em uma próxima operação de leitura, a página localizada na cache, o processador é capaz de recuperar o dado através dos *write-notices* recebidos. Essa recuperação requer não só a busca de todos os *diffs* gerados para o dado, mas também a busca da versão original do dado lida do disco armazenada na cache.

Para que um nó possa localizar a versão original do dado na *cache*, o sistema DSMIO estabelece estaticamente um nó *home* para cada dado lido do disco. Cada nó *home* é responsável por gerenciar uma parcela das páginas da cache, como pode

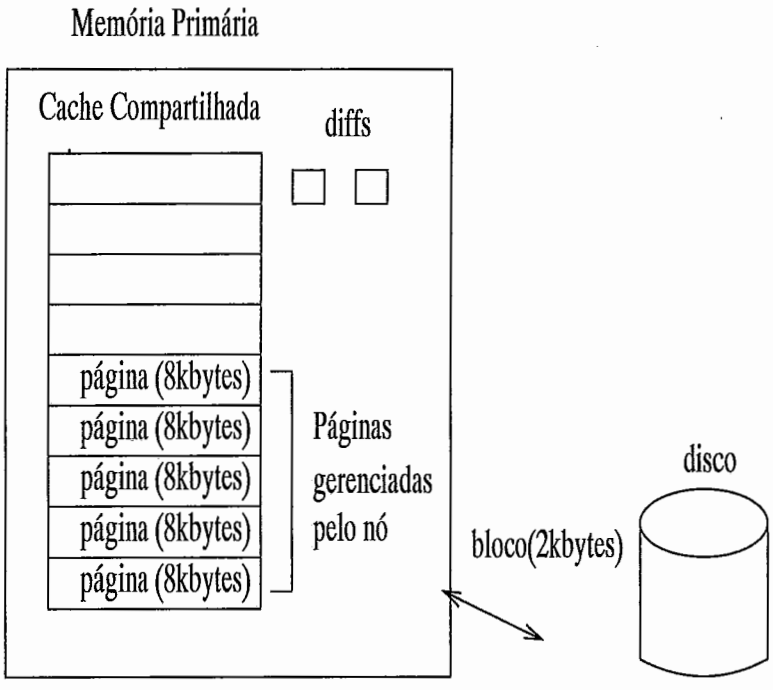

Figura 3.1: Organização da Memória de um nó Home

ser visto na figura 3.1. O nó *home* determina qual a página da cache em que o dado será armazenado e envia essa informação para o nó cliente (nó que solicita o dado ao nó *home*) quando este for receber o dado. O nó *home* lê blocos de 2kbytes do disco, para ser compatível com o tamanho do bloco lido do disco pelo SGBD GOA,(ver seção 4.2), e armazena em páginas de 8kbytes da memória primária, que é o tamanho de uma página no sistema SP/IBM, desta forma, cada página da cache compartilhada pode conter até 4 blocos lidos do disco.

3.1.2 Operações básicas do Sistema DSMIO

O sistema DSMIO provê 3 operações básicas: leitura, escrita e `flush`. Em um sistema de arquivos tradicional, uma operação de leitura ou escrita ao disco especifica como será feita a transferência de um dado localizado na memória local do nó de/para um bloco do disco. No sistema DSMIO, as operações de leitura especificam a transferência de um dado localizado no disco para páginas da cache compartilhada e da cache compartilhada para a memória local.

Em DSMIO, uma operação de escrita transfere o dado que foi modificado na memória local para as mesmas páginas compartilhadas em que este dado foi armazenado originalmente quando foi lido do disco, e então gera e armazena os *diffs* da diferença entre as páginas originais e as páginas modificadas. Neste sistema, os

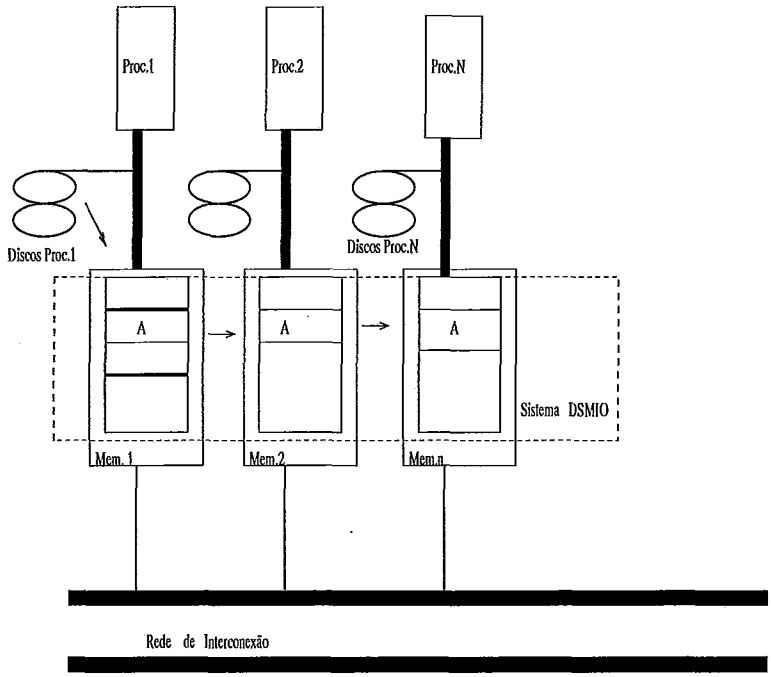

Figura 3.2: Operação de Leitura DSMIO de forma simplificada

dados são transferidos para o disco apenas em uma operação de *flush*.

Uma operação de *flush* é executada para remover páginas da cache. Ela é executada quando não há mais páginas disponíveis na área de cache gerenciada por um nó *home*, ou quando o espaço ocupado pelos *diffs* na memória local de um nó do sistema atinge um valor crítico, e ao término da execução do processo para armazenar no disco os dados que foram modificados.

As operações de sincronização utilizadas para garantir acessos ao disco com exclusão mútua são idênticas às operações oferecidas por TreadMarks para garantir exclusão mútua de acesso à memória, i.e., *lock/unlock*.

Operação de Leitura

A figura 3.2 apresenta uma forma simplificada do funcionamento de uma operação de leitura DSMIO. Nesta figura, o processador 2 e o processador N solicitam uma cópia de dados pertencentes a página A para o seu respectivo servidor, no caso o processador 1.

A figura 3.3 apresenta o algoritmo da operação de operação de leitura DSMIO. Primeiro o nó P2 verifica se a página correspondente ao dado sendo solicitado está presente localmente, i.e., se o processador já executou um pedido da página ao nó *home*.

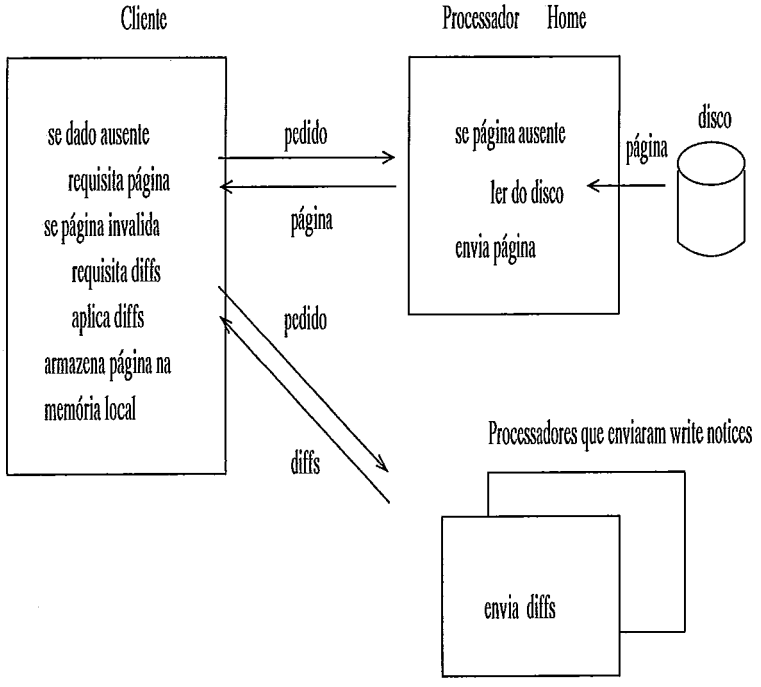

Figura 3.3: Operação de Leitura

Caso não esteja presente localmente ele deve requisitar uma cópia da página a seu respectivo nó *home*. O nó *home*, por sua vez, ao receber um pedido da página também verifica se ela está presente localmente, porque caso não esteja ele executará uma operação busca no disco para armazená-la na cache e então torná-la presente localmente. Em seguida, a página é enviada ao nó cliente. Depois que o nó cliente recebeu a página do nó home, ele verifica se a sua respectiva página compartilhada local está inválida, ou seja, se ela recebeu invalidações por causa de *write notices*. Neste caso, os respectivos *diffs* são solicitados aos nós que geraram os *write notices*. Os *diffs* recebidos são aplicados às páginas. Por fim, o dado é transferido para a área de memória local especificada pela aplicação na operação de leitura.

Operação de Escrita

A figura 3.4 apresenta uma forma simplificada do funcionamento de um operação de escrita ao disco. No caso, os processadores P2 e P3 executam primeiro uma operação de leitura DSMIO no qual recebem a página A do processador servidor P1. A seguir o processador P2 executa uma operação de escrita DSMIO na qual um dado da página A é modificado, transformando a página em uma página modificada A". A operação de escrita DSMIO faz com que apenas a diferença entre a página antiga A e a página modificada A", i.e., o *diff*, seja armazenada na memória local

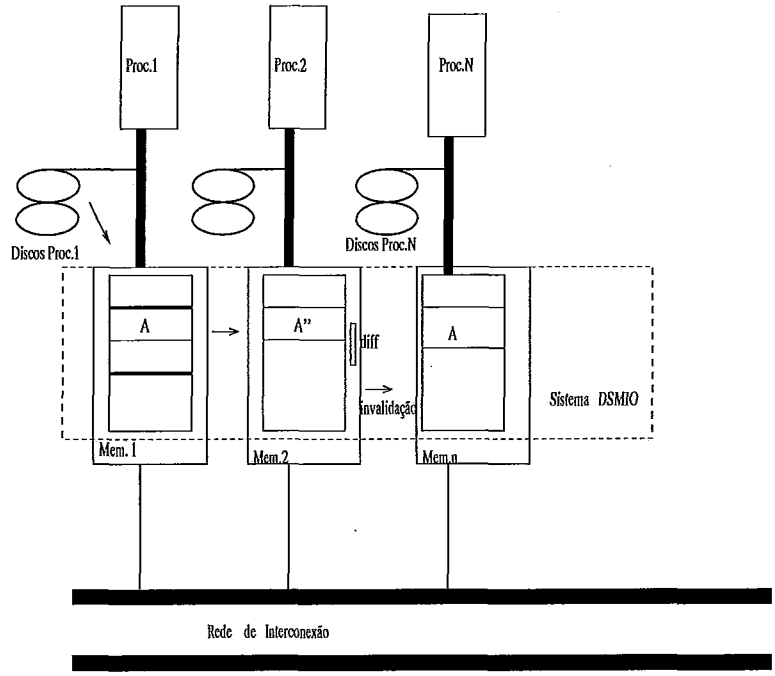

Figura 3.4: Operação de escrita DSMIO de forma simplificada

do processador P2.

No próximo pedido de aquisição de *lock* pelo processador P3, ele irá receber do processador P2 apenas a informação para ele invalidar da página A, segundo o protocolo de *Treadmarks*.

A figura 3.5 apresenta o algoritmo da operação de escrita DSMIO. Numa operação de escrita, inicialmente o sistema executa uma operação de leitura DSMIO para requisitar a página ao nó *home*, caso ela não esteja "presente localmente", e/ou receber os seus respectivos *diffs*, caso a página esteja inválida.

Estando a página presente localmente, antes de transferir o dado modificado da memória local para a cache, um *twin* é criado para a(s) página(s) em que o dado está contido. Em seguida o sistema pode então transferir o dado modificado localmente para a sua(s) respectivas páginas na memória *cache* compartilhada e criar seus respectivos *diffs*. Após a criação dos *diffs*, a área de *twin* é liberada, assim como a área física alocada para a página compartilhada se torna disponível para ser reutilizada pelo sistema operacional e, por fim, um *write notice* é gerado para esta modificação.

As estruturas de *diff* e de *write notice* geradas pela operação de escrita são incorporadas ao intervalo corrente do processador, de forma a não alterar a estrutura de intervalos da ordem parcial *hb1* conforme descrito no capítulo anterior.

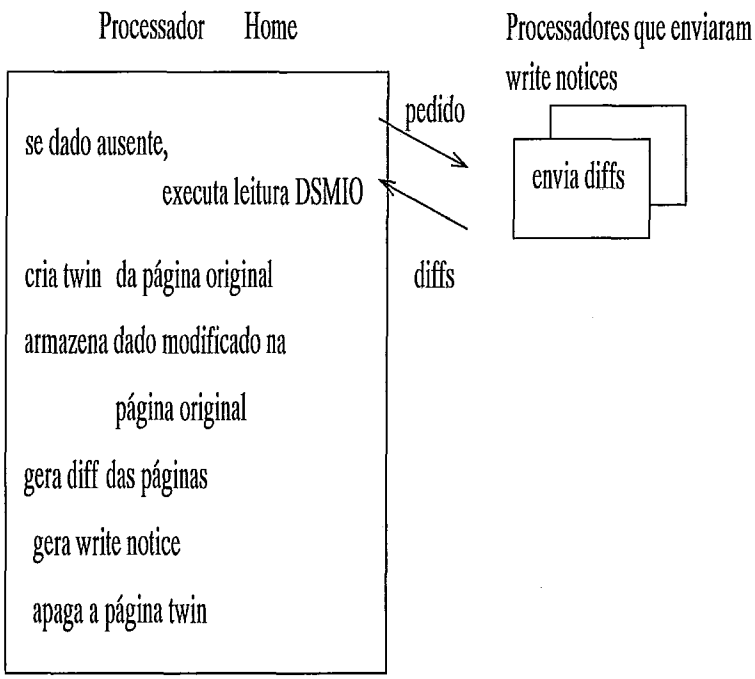

Figura 3.5: Operação de Escrita

O sistema DSMIO possui um mecanismo que evita os *overheads* relativos à criação de *diffs* quando uma página não está sendo efetivamente compartilhada. Um nó *home* controla os pedidos realizados para cada página e mantém os nós clientes informados a respeito do grau de compartilhamento das páginas. Uma operação de escrita em DSMIO, então, se adapta dinâmicamente de acordo com o grau de compartilhamento do dado. Quando a página não é efetivamente compartilhada, o sistema não cria *diffs* e envia a página modificada integralmente ao nó *home* logo após sua atualização.

Operação de Flush

Há dois tipos de operações de *flush* no sistema DSMIO: *flush* parcial e *flush* total. Uma operação de *flush* parcial é realizada quando há falta de espaço na *cache* ou na área de *diffs*. Uma operação de *flush* total é sempre executada ao término da execução da aplicação para que os dados que foram modificados na *cache* sejam armazenados de volta no disco.

A figura 3.6 apresenta uma forma simplificada do funcionamento da operação de *flush*. No caso, a operação de *flush* foi executada, apenas o processador P2 tinha executado modificações nos dados e tinha *diffs* armazenados na sua memória local. O processador P2 envia os diffs para o processador P1 que, após a recuperação das

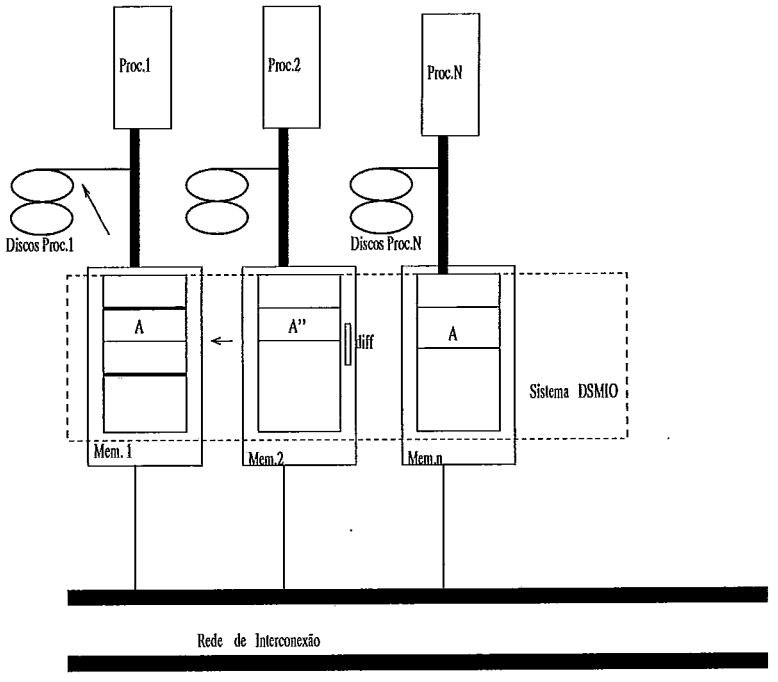

Figura 3.6: Operação de Flush DSMIO de forma simplificada

páginas, escreve de volta no disco.

As figuras 3.7 e 3.8 apresentam respectivamente os algoritmos das operações de *flush* parcial e de *flush* total. Quando uma operação de *flush* parcial (figura 3.7) é disparada por falta de espaço na *cache*, ou por falta de espaço na área de *diffs* o nó *home* envia um pedido de *flush* parcial para todos os nós clientes, para receber os respectivos *write notices* e *diffs* das páginas, pertencentes a área de memória compartilhada gerenciada pelo nó *home* geradas até o presente intervalo. Ao receber as informações dos nós clientes, o nó *home* irá percorrer cada uma das páginas inválidas para que elas sejam recuperadas e escritas de volta no disco. E alterar o estado das suas respectivas páginas compartilhadas para "não presente localmente".

Após o envio para o nó *home* dos *write notices* e de seus respectivos *diffs* os nós clientes os removem da sua área local e modificam o estado das páginas compartilhadas correspondentes ao nó *home* que acaba de executar a operação de *Flush* para "não presente localmente".

Numa operação de *flush* total (figura 3.8), cada nó *home* solicita todos os *diffs* gerados para suas páginas e os aplica para recuperar a página. Estando todas as páginas recuperadas, elas são escritas de volta no disco. Atualmente, o sistema DSMIO possui apenas a operação de *flush* total. As outras operações de *flush* ainda estão em desenvolvimento.

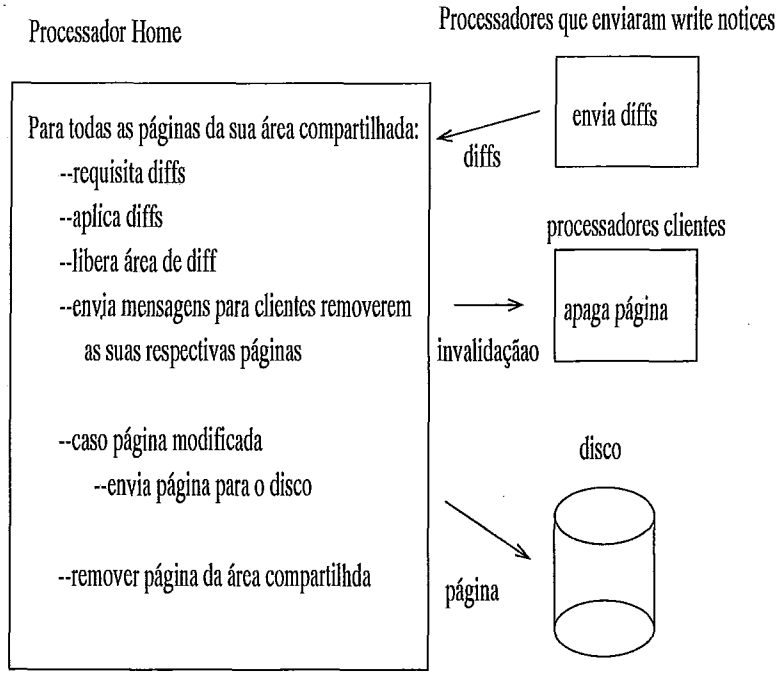

Figura 3.7: Operação de *flush* parcial.

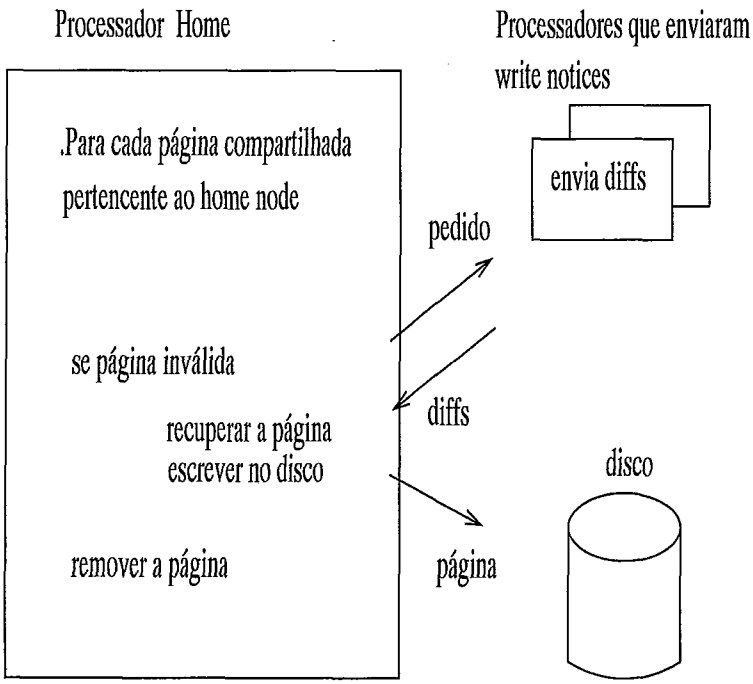

Figura 3.8: Operação de *flush* total.

3.1.3 Falhas de Acesso

De forma diferente do sistema *TreadMarks*, o sistema DSMIO inicialmente mantém todas as páginas do sistema válidas e desprotegidas para as interrupções de falhas de acesso. Isto é feito para permitir que os nós *homes* possam transferir os dados do disco para a *cache* sem causar interrupções no sistema *TreadMarks*. O sistema DSMIO tem como objetivo gerar *diffs* apenas nas operações de escrita a dados compartilhados.

Um nó *home* pode receber um pedido de leitura de uma página de um nó cliente através de um pedido de leitura remota, ou por um pedido local, executado pela operação de leitura *Tmk-DSMIO-read*.

Se a página se encontrar "localmente presente" na cache do nó *home*, ou uma cópia é transferida para o nó cliente ou o dado é transferido para a memória local.

Se a página sendo solicitada não se encontra "localmente presente" na cache, o nó *home* lê a página do disco e a transfere para uma página da memória compartilhada gerenciada por ele. A seguir, ou ele envia uma cópia da página para o nó cliente, ou envia o dado para a memória local da aplicação.

Antes do término da operação a página compartilhada passa a ter o estado de "presente localmente" e continua desprotegida para os acessos de leitura e escrita.

Um nó cliente, ao receber uma página do nó *home* recebe também a informação sobre qual a localização da página da *cache* em que a página deve ser escrita. Uma vez armazenada a página, o dado solicitado é transferido para a memória local e a página escrita continua desprotegida para acessos de leitura e de escrita.

3.1.4 Programação com DSMIO

Programas escritos para o sistema DSMIO, assim como programas *TreadMarks*, seguem o estilo convencional SPMD de programação no modelo de memória compartilhada, utilizando processos para expressar o paralelismo e primitivas de *lock/unlock* e barreira para sincronização. O sistema DSMIO provê uma extensão da biblioteca de funções do sistema TreadMarks para permitir que uma aplicação possa, além de fazer as chamadas tradicionais de escrita e leitura ao disco, fazer também chamadas de escrita e leitura à cache de disco compartilhada. As primitivas `Tmk-DSMIO-read` e `Tmk-DSMIO-write` realizam essas operações.

A inicialização dos processos paralelos é realizada por um único processo (pro-

cesso pai que executa no processador 0), através da primitiva `Tmk-startup`. É inicializado um único processo em cada processador. Cada processo paralelo, então, inicializa as estruturas internas do sistema *Treadmarks* e aloca 32MB de sua memória privativa para armazenar os dados compartilhados. O processo pai é responsável por alocar cada estrutura compartilhada da aplicação nessa área assim como a área da cache compartilhada através da primitiva `Tmk-malloc`. Em seguida ele distribui para os outros processadores os endereços das estruturas alocadas através da primitiva `Tmk-distribute`. Depois que a distribuição dos endereços estiver completa, todos os processos (inclusive o processo pai) executam o mesmo código, realizando sua parcela do trabalho e se comunicando através das primitivas de sincronização.

Ao término da execução, a aplicação deve chamar a primitiva `Tmk-flush` para executar uma operação de *flush* total em cada nó, que também é executada pelo nó pai.

Ao final, a aplicação tem que chamar a primitiva *Tmk-exit* para finalizar os processos inicializados nos nós de cada processador e para coletar seus respectivos dados de saída, caso desejado para serem utilizados na geração de dados de estatísticas.

Capítulo 4

Arquiteturas Paralelas de SGBDs

Esse capítulo tem como objetivo descrever os sistemas utilizados para avaliar o desempenho do sistema DSMIO.Inicialmente apresentaremos as arquiteturas paralelas de Sistemas Gerentes de Banco de Dados, SGBD, mais conhecidas. Em seguida apresentaremos o SGBDOO GOA, desenvolvido pela COPPE/UFRJ, o GOA [15], utilizado para a avaliação de desempenho do sistema DSMIO e as aplicações teste, obtidoas do *Benchmark 007* desenvolvido pela Universidade de Wisconsin [4].

4.1 Sistemas Paralelos de Banco de Dados

Sistemas Paralelos de Banco de Dados (SPBD) surgiram como forma de se fornecer um maior desempenho aos SGBD. De forma simplificada, um SPBD pode ser definido como um SGBD implementado em uma máquina com múltiplos processadores, ou seja, um computador com capacidade de processamento paralelo., onde um SPBD é igual a um SGBD paralelo somado a Banco de Dados.

Existe uma outra arquitetura, a cliente-servidor, que não é paralela, mas que é também utilizada com o objetivo de aumentar o desempenho de SGBDs. Essa arquitetura distribui o trabalho em processamentos concorrentes. Desta forma, por exemplo, enquanto um processador cliente analisa os resultados de uma consulta em sua estação de trabalho, um processador servidor está disponivel para o atendimento às solicitações de outros clientes [15, 5].

A organização da memória (principal e secundária) é um dos parâmetros da arquitetura paralela hospedeira que tem influência decisiva no projeto de SPBDs [61].

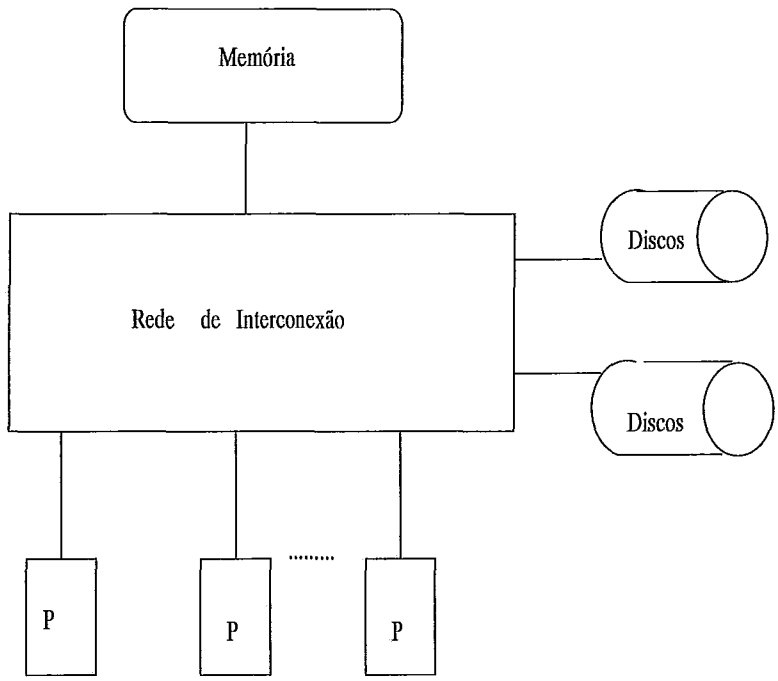

Figura 4.1: Arquitetura de Memória Compartilhada

4.1.1 Arquitetura de Memória Compartilhada

Neste tipo de arquitetura, como pode ser visto na figura 4.1 todos os processadores **P** têm acesso a qualquer módulo de memória através de uma rápida rede de interconexão. Este modelo de memória apresenta duas grandes vantagens: simplicidade e balanceamento de carga. Uma vez que o dicionário e as informações de controle podem ser recuperadas por todos os processadores, os procedimentos para gerenciamento do banco são similares aos empregados em sistemas com monoprocessamento. O balanceamento de carga é excelente, já que o sistema associa as tarefas aos processadores em tempo de execução baseado na carga atual dos mesmos.

Como desvantagens podem ser citadas o custo, a expansibilidade limitada e a baixa disponibilidade. Os custos são altos devido à complexidade da rede de interconexão, pois é necessário que se ligue cada processador a cada módulo de memória e a cada disco do sistema. Acessos conflitantes à memória compartilhada podem causar interferência entre os processadores limitando o crescimento do sistema. Segundo [17], a rede de interconexão deve ter uma grande largura de banda sendo portanto difícil construir redes que apresentem escalabilidade. Além disso, uma vez que o espaço de memória é dividido por todos os processadores, uma falha neste

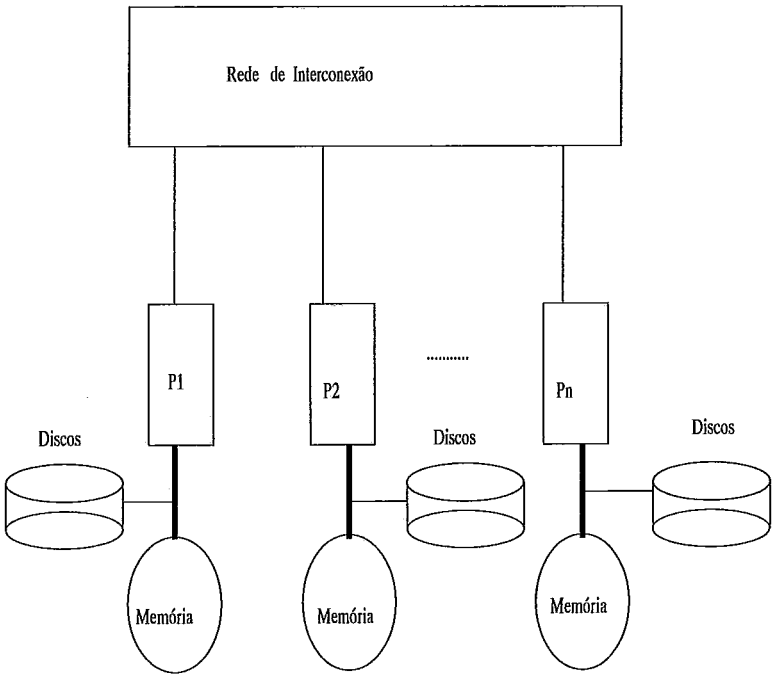

Figura 4.2: Arquitetura de Memória distribuída

componente pode afetar todos os processadores.

4.1.2 Arquitetura de Memória Distribuída

Na arquitetura de memória distribuída, como pode ser visto na figura 4.2, cada processador **P** tem acesso exclusivo à sua memória e à(s) sua(s) unidade(s) de disco. Cada nó (processador + memória + disco) pode ser visto como um SGBD local da mesma forma que os bancos de dados distribuídos. Devido a esta similaridade, muitas das soluções projetadas para SGBDs distribuídos tais como, fragmentação, gerenciamento distribuído de transações e processamento de consultas distribuído podem ser reutilizadas neste modelo [18].

As principais vantagens da memória distribuída são o baixo custo aliado a uma alta expansibilidade e disponibilidade. Ao permitir um crescimento incremental, é possível atender a um grande número de processadores, minimizando interferências através da redução de recursos compartilhados. A replicação de dados entre os nós permite que seja obtido um alto grau de disponibilidade do sistema [17].

O maior problema desta arquitetura é o balanceamento de carga, o qual pode degradar consideravelmente o desempenho do sistema.

Segundo Valduriez [19], para pequenas configurações (menos de 20 processadores) a memória compartilhada pode oferecer o melhor balanceamento de carga.

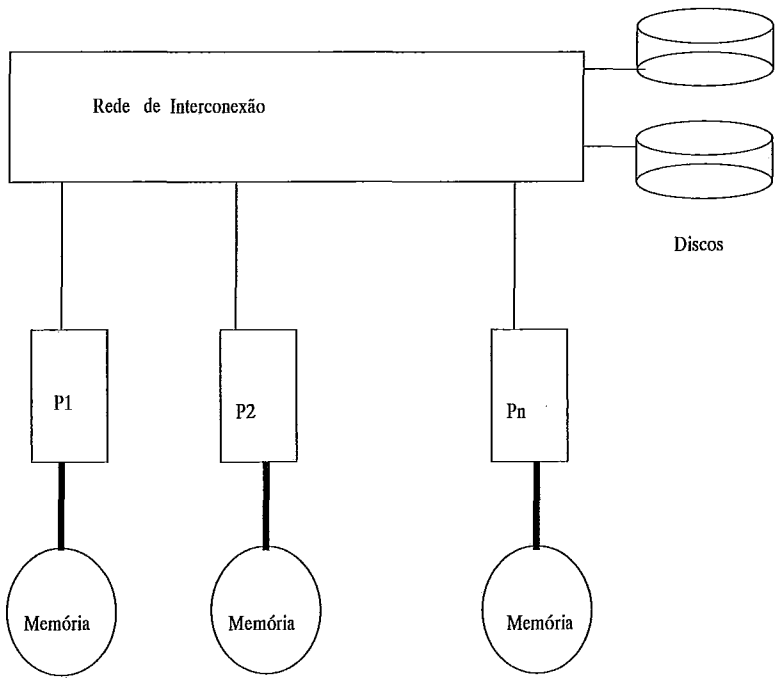

Figura 4.3: Arquitetura de Disco Compartilhado

Entretanto, as arquiteturas de disco compartilhado e as de memória distribuída superam o modelo de memória compartilhada tanto em expansibilidade como em disponibilidade sendo que a arquitetura de memória distribuída consegue um grau de expansão muito superior às outras duas arquiteturas.

4.1.3 Arquitetura de Disco Compartilhado

No modelo de arquitetura de disco compartilhado, como pode ser visto na figura 4.3, cada processador possui uma memória exclusiva e tem acesso a qualquer unidade de disco através da rede de interconexão. Segundo [19], as principais vantagens desta arquitetura são: custo, expansibilidade, balanceamento de carga, disponibilidade e facilidade para migrar de um sistema monoprocessado. O custo da rede de interconexão é bem inferior ao modelo de memória compartilhada já que a tecnologia de barramento padrão pode ser utilizada. Como cada processador tem a si associado um módulo de memória, o problema de interferência pode ser minimizado permitindo uma expansibilidade maior (o da ordem de centenas de processadores). O balanceamento de carga pode ser tão bom quanto o da memória compartilhada e falhas nos módulos de memória podem ser isoladas dos outros nós possibilitando assim uma maior disponibilidade do sistema.

O acesso aos discos compartilhados representa também potenciais problemas de

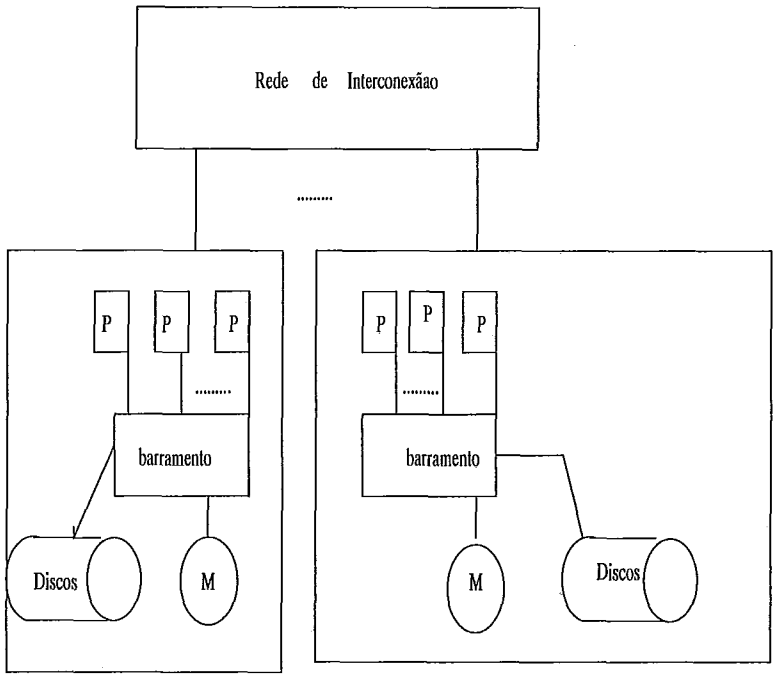

Figura 4.4: Arquitetura Hierárquica

gargalo de E/S. Desta forma, segundo [17], este tipo de arquitetura é mais adequada para aplicações do tipo *read-only.*

4.1.4 Arquitetura Hierárquica

As arquiteturas hierárquicas, como pode ser visto pela figura 4.4, são uma combinação das arquiteturas de memória compartilhada e de memória distribuída. A idéia é construir uma máquina de memória distribuída em cima de nós de memória compartilhada. As vantagens destas arquiteturas são evidentes. Ela combina a flexibilidade e o desempenho de sistemas de memória compartilhada com a alta expansiblidade das arquiteturas memória distribuída.

4.1.5 Arquitetura de Acesso de Memória não Uniforme

As arquiteturas de Acesso de Memória não uniforme, ou *Non Uniform Memory Acess, NUMA*, têm o mesmo objetivo de combinar a flexibilidade e a expansibilidade, de forma que os multiprocessadores *shared memory* estão migrando em direção às arquiteturas NUMA. O objetivo é fornecer o modelo de programação de memória compartilhada junto com todos os seus benefícios, em uma arquitetura paralela escalável.

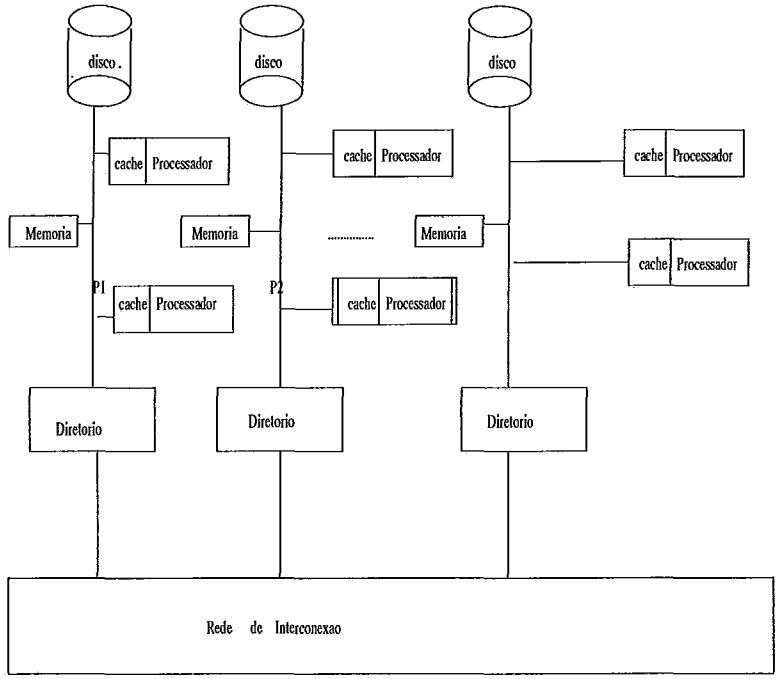

Figura 4.5: Arquitetura cc-NUMA

Duas classes de arquitetura NUMA emergiram, *Cache Coherent NUMA* (cc-NUMA) (figura 4.5) que divide estaticamente a memória principal entre os nós do sistema e *Cache Only Memory Architeture* (COMA)(figura 4.6), onde a memória de cada nó do sistema faz parte de uma grande e única memória *cache* no espaço de endereçamento compartilhado, de forma que a localização do dado é completamente desacoplada do seu endereçamento físico e o dado é automaticamente migrado ou replicado na memória principal. Devido ao fato de que a memória é compartilhada e de que a coerência dos dados é mantida por *hardware*, os acessos remotos à memória são muito eficientes.

4.1.6 Arquitetura Cliente-Servidor

A arquitetura cliente-servidor (figura 4.7) é comumente empregada em sistemas de redes de estação de trabalho. Um SGBD cliente-servidor divide os processos em dois tipos; processos clientes, que são executados nas estações de trabalho e permitem a interação com as aplicações do usuário, e processos servidores, que são tipicamente executados em máquinas servidores que permitem o acesso ao banco de dados atendendo aos pedidos dos múltiplos clientes [7].

O desenvolvimento do modelo cliente-servidor veio em decorrência dos avanços relacionados não só aos sistemas distribuídos como também com relação à pesqui-

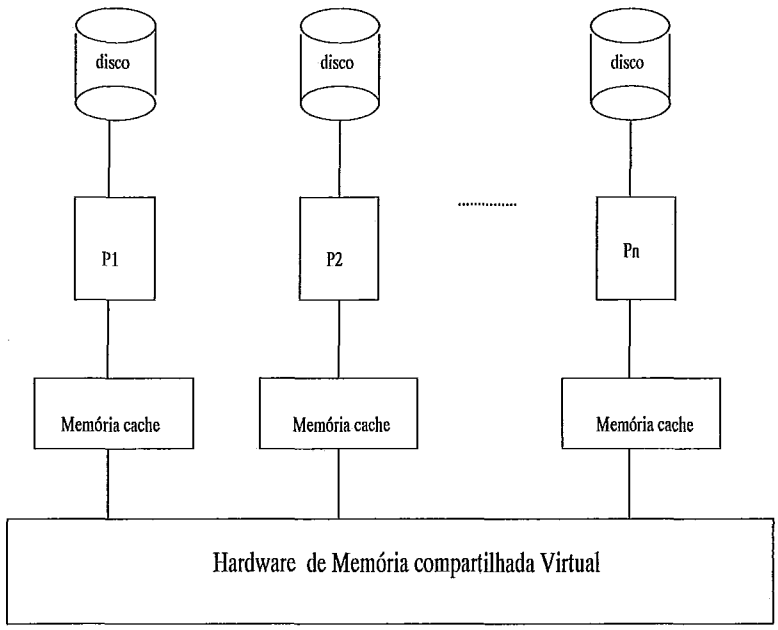

Figura 4.6: Arquitetura COMA

sa em orientação a objeto [8]. Uma arquitetura cliente-servidor pode adotar dois modelos de transferência de dados entre o cliente e o servidor. No modelo *query shipping* o cliente transmite um pedido de execução de uma ou mais operações de consulta para o servidor, e no modelo *data shipping*, o cliente solicita os dados para o servidor. A arquitetura cliente-servidor tem como objetivo trazer disponibilidade do dado ao usuário e suporte para dados complexos e compartilhados em um ambiente distribuído, de forma que tem sido largamente adotada nos sistemas de banco de dados orientados a objetos (SGBDOO).

4.2 Os sistemas Gerente de Banco de Dados Paralelos Orientados a Objeto (SGBDOO)

Os sistemas SGBDOO foram inicialmente desenvolvidos para dar suporte para aplicações de computação intensiva tal como *CAD, CAM, CASE*, entre outros. Estes sistemas adotaram o modelo *data shipping*, que é um modelo que permite que o pedido pelo processamento do dado possa ser executado no nó cliente. Com o modelo *data shipping* [12], o *software* SGBDPOO, que é executado no cliente, determina quais são os dados necessários para satisfazer um determinado pedido de uma aplicação e é necessário requisitar ao servidor caso não possa ser satisfeito localmente.

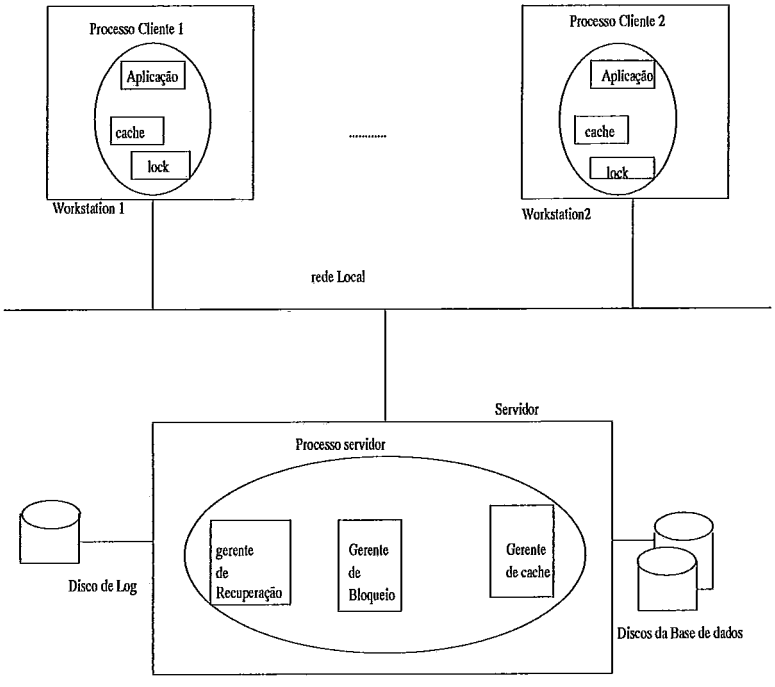

Figura 4.7: Arquitetura Cliente Servidor

São duas as vantagens do modelo *data shipping* para os SGBDPOO: primeiro, *data shipping* move o dado para perto da aplicação, de forma que permite acelerar o processo de navegação através das estuturas de dados persistentes através das interfaces dos SGBD Orientados a Objeto. Segundo, o modelo *data shiping* permite retirar grande parte da carga de trabalho do servidor, permitindo um melhor desempenho do sistema assim como possibilitando um aumento na escalabilidade. Como resultado, o modelo *data shipping* é empregado em sistemas de pesquisas tais como EXODUS [34] e SHORE, assim como em produtos comerciais tais como O2 [33]e Object Store [32].

Enquanto que o modelo *data-shipping* traz um grande benefício aos sistemas de banco de dados orientados a objeto cliente-servidor, este modelo também torna o sistema altamente suscetível à rede e à taxa de dados pedidos pelos clientes ao servidor. Para evitar estes problemas, se faz necessário um *caching* de dados nos clientes de forma a permitir que o cliente possa manter as cópias dos dados recebidos pelo servidor e, com isto, diminuir a taxa de dados pedidos.

4.3 O Gerente de Objetos Armazenados (GOA)

No final da década de 80, foi iniciado no Programa de Engenharia de Sistemas de Computação da COPPE/UFRJ o projeto TABA [15], um ambiente de desenvolvimento de *software* configurável segundo as necessidades do usuário, composto por vários módulos entre os quais um sistema de gerência de bases de dados orientado a objetos denominado GEOTABA. Como parte do GEOTABA foi desenvolvido um componente para a Gerência de Objetos Armazenados, o GOA.

O GOA possui uma arquitetura cliente-servidor de objetos. Ele pode ser acoplado a outros SGBDOOs ou a sistemas desenvolvidos em linguagens de programação orientadas a objetos que necessitem de persistência de objetos e consultas *ad-hoc*. O sistema apresenta dois componentes principais: O Gerente de Meio de Armazenamento (GMA), responsável pela organização e acesso físico aos meios de armazenamento e o Gerente de Memória de Objetos (GMO), responsável pelo gerenciamento da memória primária . Além destes dois componentes, existe o módulo que responde pelo processamento de consultas, o PC. Para o gerenciamento dos objetos em memória, é utilizado um *buffer* de páginas com tamanho de 4 Mbytes.

Com relação às estruturas de dados utilizadas, o sistema apresenta as seguintes características :

- O identificador de objeto (IDO) é físico e formado por dois componentes: o primeiro que identifica a página no disco e o segundo o deslocamento dentro da página.
- Os objetos embutem na sua representação a descrição dos atributos. Para cada atributo do objeto são armazenados o seu nome, tamanho e deslocamento para o seu valor.
- Os atributos podem ser simples, de referência ou multivalorados.
- Construtores do tipo lista e coleção. O primeiro suporta o armazenamento de objetos da classe lista e facilitam a implementação de atributos multivalorados, enquanto o segundo construtor visa o armazenamento dos IDOs dos objetos que pertencem a uma classe do tipo coleção.

O servidor executa um conjunto de operações para a manipulação de objetos e construtores, tais como alocação, modificação e remoção, e é responsável pela

avaliação de predicados no processamento das consultas. Da mesma forma que nos sistemas comerciais Itasca [22] e O2 [33], a sua linguagem de consulta tem acesso aos objetos de uma classe qualquer. Porém, a classe alvo da consulta deve ser do tipo coleção, de forma semelhante ao O2 e ao padrão ODMG [96], onde o alvo da consulta é sobre uma coleção nomeada *named set.*

4.4 O *Benchmark 007*

O Benchmark 007 [4] é uma ferramenta desenvolvida na Universidade de Wisconsin-Madison, com o intuito de possibilitar a avaliação de desempenho de bancos de dados orientados a objetos. Essa aplicação tem sido adotada como um padrão nas avaliações de SGBDOOs tanto para comparar desempenho de produtos comerciais como para comparar o comportamento de um sistema face a diferentes configurações [15] [12] [11].

Sendo assim, este *benchmark* testa, entre outras, as seguintes características de desempenho dos gerenciadores SGBDOO:

- Velocidade das diversas formas de preservar ponteiros, sejam eles relativos a dados em *cache* ou no disco, de forma dispersa ou densa.
- Eficiência em vários tipos de atualização, estas incluindo as nos campos do objetos, indexados ou não, as esparsas, as em dados no *cache*, e a criação e destruição de objetos.
- O processador de consultas em várias situações.

O 007 especifica 17 operações que podem ser divididas em categorias de operações de consultas e de operações *traversal.* Uma operação de consulta é uma operação de busca de objetos que satisfazem um determinado predicado. Uma operação *traversal* navega de objeto para objeto de forma a satisfazer um predicado complexo e pode envolver atualizações a base de dados.

Na sua implementação no GOA, foram feitas algumas adaptações no esquema de dados (figura 4.8), em função de características não implementadas no GOA e por estarem fora do escopo dos objetivos do nosso trabalho. As simplificações retiraram do modelo as classes *Module, ComplexAssembly e Manual* referentes à parte mais alta da hierarquia. Entretanto, as classes que restaram permitem realizar satisfatoriamente os testes com distribuição e paralelismo uma vez que estas classes

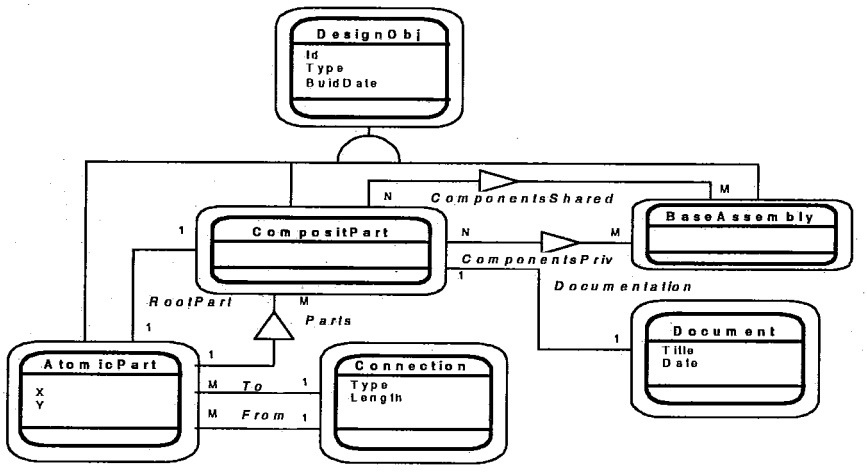

Figura 4.8: A implementação da base 007 no GOA

são altamente relacionadas, possuem populações distintas e permitem aproveitar uma situação bastante genérica.

Do seu esquema original, foram mantidas as classes *DesignObject, BaseAssembly, CompositPart, Document, AtomicPart e Connections.* A figura 4.8 mostra o diagrama de classes do 007 adaptado segundo os critérios apresentados.

Segundo seus autores o componente chave do 007 são as *CompositPart.* Cada *CompositPart* tem associado um *Document* e é formado por um conjunto de 200 *AtomicParts* que se conectam entre si através das *Connections.* O número de *Connections* por *AtomicPart* foi fixado em 3. Uma *BaseAssembly* é formada por várias *CompositParts* divididas em *ComponentsShared e ComponentsPriv.*

O 007 na sua forma original admite três tamanhos de base: pequena, média, e grande. A base pequena tem por objetivo permitir que todos os objetos caibam em memória, e embora apresente bons resultados com o paralelismo [3], não é alvo do processamento paralelo. Já no caso da base grande, segundo os autores, seria destinada a testar versões multi-usuário com objetos compartilhados. Entretanto não tem sido utilizada em nenhum dos trabalhos encontrados na literatura [21], e além disto o estudo sendo desenvolvido na COPPE é para um ambiente mono-usuário. Portanto, nesta tese a implementação foi feita na base de tamanho médio, que tem sido representativa e utilizada pelos fabricantes para análise de desempenho.

Tabela 4.1: Número de objetos por classe.

Classe	População
BaseAssembly	200
CompositParts	500
Documents	500
AtomicParts	100.000
Connections	300.000
Design Object	1

As populações de objetos em cada classe são mostradas na tabela 4.1.

Capítulo 5

Avaliação do desempenho do Sistema DSMIO

Neste capítulo iremos avaliar o desempenho dos mecanismos de *Software DSM* do sistema DSMIO em relação a modelos de consistência de cache menos relaxados e em relação aos mecanismos de *diffs* fornecidos pelo sistema *TreadMarks*.

5.1 GOA+NFS versus GOA+DSMIO

Neste primeiro teste, iremos avaliar o desempenho do sistema DSMIO em relação a um sistema SGBD tradicional [72], cuja manutenção da coerência dos dados compartilhados é mantida no disco com um modelo de consistência seqüencial [89].

Para avaliar o desempenho do sistema DSMIO, utilizamos um multicomputador IBM SP com 4 nós de processamento, onde cada nó é composto de um processador Power2 com 256 Mbytes de memória e um disco local de 2Gbytes. Neste teste nós utilizamos apenas 4 processadores por estarmos acessando um único disco via NFS. Este estudo mantém o número de processadores constante e varia a taxa de objetos compartilhados.

Os nós estavam dedicados a esta aplicação. A instrumentação necessária para coletar os tempos medidos afeta o desempenho do sistema de forma negligível.

Para avaliarmos o estudo de desempenho utilizamos o sistema SGBD GOA [15, 2]. que possui uma cache de 4Mbytes para armazenar em memória os objetos lidos do disco.

Para gerar um sistema SGBD paralelo tradicional, no caso o GOA+NFS, utilizamos o sistema TreadMarks para criar um processo GOA por processador e para fornecer operações de sincronização entre processos, como *lock/unlock* e barreira.

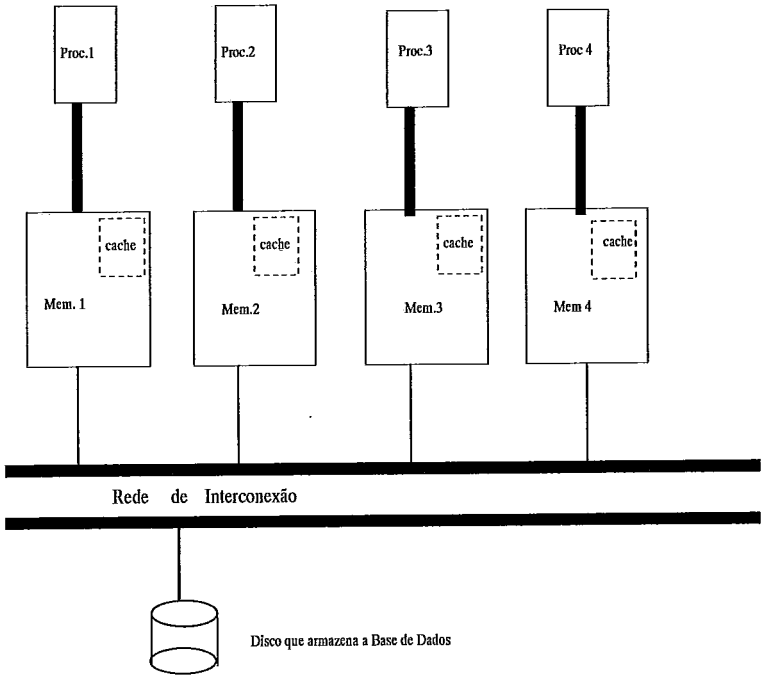

Figura 5.1: Sistema SGBD paralelo GOA acessando a base de dados via NFS

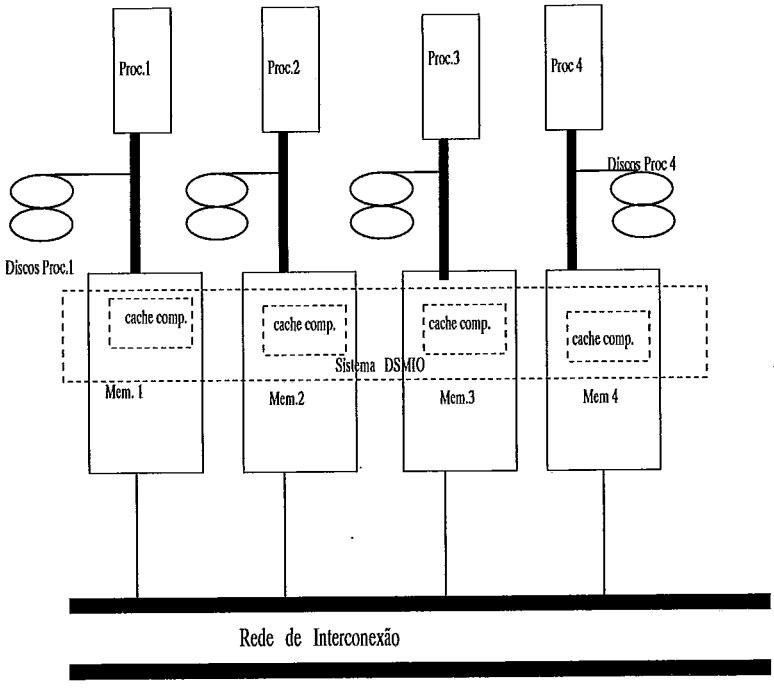

Figura 5.2: Sistema SGBD paralelo GOA acessando a base de dados via Sistema DSMIO

Os teste comparam o desempenho de dois sistemas SGBD paralelos, ambos desenvovidos através da paralelização do GOA com utilizando o sistema *TreadMarks*:

- O GOA+NFS (figura 5.1) cujos nós acessam uma base de dados comum através do sistema *Network File System* (NFS) [72] e utilizam operações de *fsync* para manter a consistência dos dados no disco segundo um modelo de consistência seqüencial.

- O SGBD paralelo GOA+DSMIO, desenvolvido através do sistema *Treadmarks* (figura 5.2) onde as caches locais de cada processo SGBD GOA executando em cada um dos processadores são substituídas pela cache compartilhada gerenciada pelo sistema DSMIO, descrito no capítulo 3. A base de dados é fragmentada entre os 4 processadores do sistema de maneira *round-robin*.

Função *fsync*

A função *fsync* é uma função oferecida pelo sistema operacional UNIX que tem como finalidade escrever todos os dados que estão armazenados no *buffer* de E/S de volta para um determinado arquivo no disco.

Esta função é utilizada por diversos sistemas de arquivos distribuídos após uma operação de escrita a um dado compartilhado. Esta função escreve no disco todos os dados armazenados no *buffer* de E/S mesmo que apenas uma página do *buffer* tenha sido atualizada.

Aplicação utilizada para o teste

Implementamos aplicações de consultas de suporte à decisão pertencentes ao *Benchmark 007* [4]. É importante lembrarmos aqui que as aplicações do *Benchmark 007* representam uma grande classe de aplicações Orientado a Objeto, tais como *CAD, CAM, CASE e SIG*. E que as três aplicações de atualização especificadas pelo *Behchmark*, as *Traversals*, possuem o mesmo comportamento de E/S. Desenvolvemos a consulta chamada de **T3** como um exemplo de uma operação de *traversal*, representativa em termos de E/S das aplicações *traversals* na qual *CompositeParts* são visitadas e, para cada uma delas, uma busca em profundidade é realizada no grafo de *AtomicParts*, e que durante a busca, todos os objetos *AtomicPart* que satisfazem o predicado são atualizados. Desta forma avaliamos relacionamentos

complexos com operações fundamentais em SGBDOO assim como as operações de junção nos SGBD relacionais.

No sistema GOA+NFS todos os processadores acessam uma base de dados localizada remotamente via o sistema NFS. No sistema DSMIO, a base de dados é fragmentada entre os discos dos processadores do sistema de forma circular (*round-robin*). Nos dois sistemas, cada processador recebe uma faixa de objetos *Assembly-Part* para ser percorrido pela *traversal* **T3**. No sistema GOA+DSMIO a fragmentação é tal que 50% dos objetos são localizados na sua base e 50% dos objetos são acessados remotamente como pedidos às bases dos processadores do sistema.

As consultas do *Benchmark 007* foram especificadas visando a processos seqüenciais. De forma a avaliar nosso sistema paralelo, as consultas *traversals* foram paralelizadas segundo o modo de programação SPMD, com operações de sincronização para evitar acessos conflitantes aos objetos compartilhados. Nos nossos experimentos, utilizamos a base de tamanho médio, 100 Mbytes, que é a base adotada como padrão para estudos em SGBD e também por ser a base 007 desenvolvida para o sistema GOA.

É importante observar que embora estejamos utilizando uma única classe de *traversal*, ela é bastante representativa para avaliação do sistema DSMIO em operações de leitura e escrita ao disco em situações de compartilhamento de dados. Isto é devido às caracteristicas das aplicações do *Benchmark 007* que representa uma classe de aplicações conforme apresentado em detalhes na seção 4.3. Além disso, as três classes de *traversals* do *benchmark* apresentam comportamento de E/S similar a **T3**. A aplicação **T3** possui localidade temporal e espacial. Cada registro lido do disco é acessado diversas vezes pela aplicação para realizar a busca seguida da atualização dos objetos. Uma vez que estes objetos terminam de ser atualizados, o próximo registro a ser lido está localizado próximo ao que já foi lido, conforme ocorre com freqüência no armazenamento de dados em SGBDs. Neste último caso, a existência de um sistema de *prefetching* no sistema de *I/O buffer* pode reduzir os acessos ao disco e alterar significativamente o desempenho da aplicação. O que foi o caso nos nossos testes: o tempo de leitura de uma página no disco mostrou-se ser da ordem do tempo de cópia da página do *buffer* do sistema de E/S para a memória local.

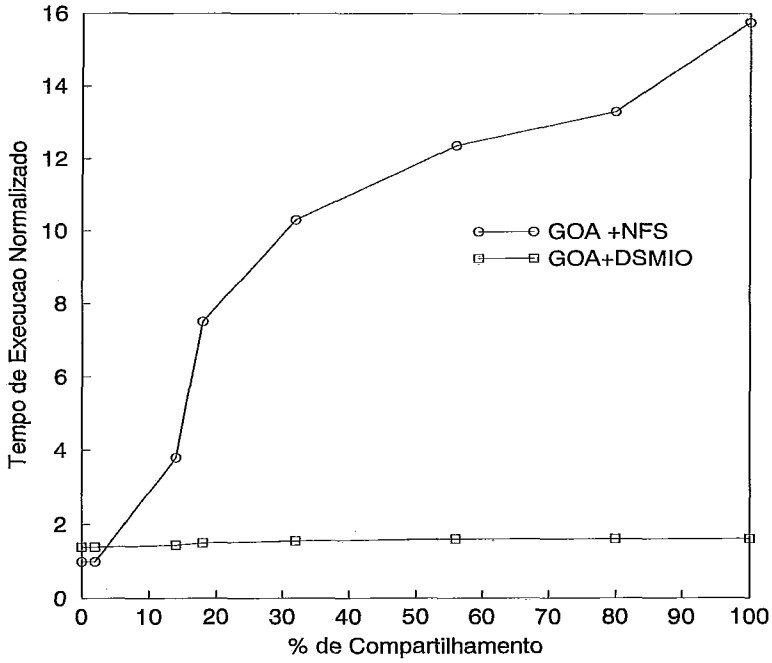

Figura 5.3: Desempenho em função do compartilhamento de dados

5.1.1 Análise de Resultados

Nesta seção comparamos o desempenho dos sistemas GOA+NFS *versus* GOA+DSMIO. Nos concentramos apenas nos aspectos de coerência de cache.

Na figura 5.3, apresentamos o desempenho de **T3** em função da porcentagem dos objetos da classe *CompositeParts* que são compartilhados. Cada conjunto de objetos compartilhados é selecionado e atualizado por todos os processadores. De forma que 0/cada objeto *AssemblyPart* será percorrido por apenas 1 processador do sistema e 100% de compartilhamento siginifica que todos os objetos *AssemblyPart* serão percorridos por todos os 4 processadores do sistema. Os tempos de execução estão normalizados em relação ao tempo do sistema GOA+NFS com 0% de compartilhamento.

Observamos na figura que, a medida que a porcentagem dos dados compartilhados aumenta, o tempo de execução de GOA+NFS aumenta linearmente, enquanto que o tempo de execução de GOA+DSMIO aumenta de forma desprezível. Isto se deve ao fato de que no sistema GOA+DSMIO, um nó ao modificar um dado compartilhado, tem que receber antes e recuperar os *diffs* correspondentes a este dado, enquanto que no sistema GOA+NFS , este nó para executar a mesma operação tem que ter antes a última versão do dado escrita no disco para então ler o dado do

disco (i.e. executar uma operação de *fsync*). Desta forma, a recuperação de um dado compartilhado no sistema GOA+DSMIO é aproximadamente 2% do tempo de recuperação de um dado compartilhado em um sistema GOA+NFS.

Esses resultados preliminares mostram que GOA+DSMIO obtém resultados bastante diferentes dos de GOA+NFS. Essa diferença varia de 40% de perda a 98% de ganho no tempo de execução. Quando nenhum dos dados é compartilhado, GOA+DSMIO é mais lento do que GOA+NFS devido à quantidade de operações de leitura e escrita em disco, as quais são 25% e 75% mais lentas, respectivamente, em GOA+DSMIO. As operações de leitura são mais lentas nesse caso, uma vez que executam um excesso de operações de troca de mensagem entre os nós clientes e os nós *home*. Esse excesso se deve à forma circular com a qual associamos dados a nós *home*. Como essa associação não leva em conta os processos que realmente acessam os dados, um processo precisa de um par de mensagens extras para se comunicar com o *home* para requisitar um dado.

Ainda considerando a situação onde não há compartilhamento efetivo de dados, as operações de escrita são mais lentas uma vez que pedidos de escrita são transformados por DSMIO em criações de *diff*, independente de existir compartilhamento do dado escrito. A criação de *diff* é uma operação custosa, respondendo por 50% do *overhead* gerado em operações de escrita. Além da criação de *diffs*, as operações de *flush* total e geração de *write notices* respondem por 12% e 25%, respectivamente, dos *overheads* de escrita. Outras operações, tais como gerenciamento de *twins* e atendimento de interrupções geradas por pedidos externos, respondem pelo resto do *overhead*.

Quando existe compartilhamento de dados, mesmo que restrito, o sistema GOA+DSMIO passa a apresentar melhor desempenho do que o sistema GOA+NFS. Isso mostra que os *overheads* envolvidos no gerenciamento de *diffs* em GOA+DSMIO são suplantados pelos ganhos que esse sistema obtém através da eliminação de acessos a disco. No caso extremo, onde todos os dados são compartilhados, por exemplo, observamos que GOA+DSMIO apresenta uma redução de 98% no tempo de execução em relação a GOA+NFS. Essa redução se deve à eliminação das operações de escrita no disco, necessárias para garantir a coerência dos dados, cujos gastos s ao gerados pelas operações de *fsync* e pela centralização do acesso ao disco. Em GOA+DSMIO essas operações não são necessárias, uma vez que as caches locais formam uma cache compartilhada.

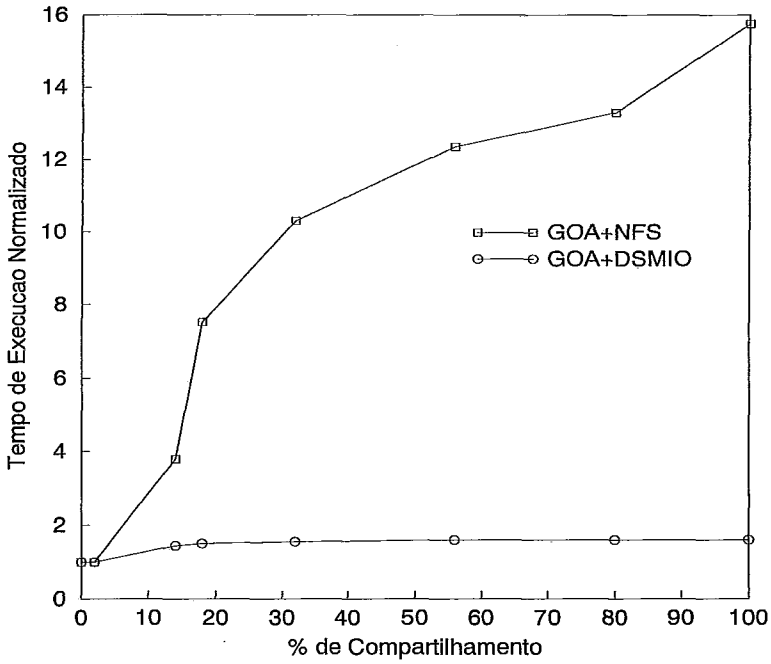

Figura 5.4: Desempenho do GOA adaptativo em função do compartilhamento de dados.

5.1.2 GOA+DSMIO Adaptativo para Acessos de Escrita

Devido aos resultados acima, alteramos o sistema DSMIO para torná-lo adaptativo ao grau de compartilhamento dos dados. Desta forma cada nó servidor gera uma informação sobre o grau de compartilhamento das páginas da *cache* compartilhada gerenciadas por eles e transmite esta informação aos seus nós clientes. Os nós clientes, por sua vez, em uma operação de escrita, passam a criar *diffs* apenas quando o dado é compartilhado por mais de um nó cliente, caso contrário, a página atualizada é enviada para o seu respectivo nó *home*. Isto melhorou muito o desempenho para operações de escrita sem compartilhamento de dados, conforme pode ser visto na figura 5.4.

Note que, quando nenhum dos dados é compartilhado, GOA+DSMIO é 25% mais lento do que GOA+NFS devido às operações de leitura em disco. No GOA+NFS o dado é solicitado diretamente do disco, enquanto que no sistema GOA+DSMIO o dado é distribuído pelos nós *home* e precisam ser solicitados para eles, que por sua vez lêem o dado do disco.

Estudos anteriores [2] comprovam que um ambiente NFS apresenta um bom desempenho para o acesso a dados de um mesmo disco até no máximo 4 processadores, e que, para um número maior de processadores, a contenção dos dados no

barramento torna o desempenho muito ruim. Enquanto que no sistema DSMIO, o sistema de divisão da base de dados entre os processadores do sistema, os nós *home*, diminui substancialmente a contenção no acesso aos dados e permite que um número muito maior de processadores possa ser integrado ao sistema.

5.1.3 Conclusões

Os resultados apresentados acima comprovam a eficiência do sistema DSMIO em trazer a coerência dos dados armazenados em disco para o nível da memória principal, implementando uma *cache* compartilhada com modelo de consistência relaxado.

Para comprovarmos os ganhos gerados pela implementação de uma arquitetura de nós *home* com relação a uma arquitetura NFS, deveríamos apresentar os mesmos testes para um número maior de processadores, 8 por exemplo. Neste caso, GOA+NFS apresentaria um desempenho muito ruim mesmo para pequenas porcentagens de compartilhamento de dados. Este assunto foi devidamente abordado em estudos anteriores [2, 3].

5.2 GOA+RC versus GOA+DSMIO

Vários sistemas de arquivos distribuídos apresentam propostas para trazer a coerência dos dados armazenados em disco para o nível da memória principal. Em termos do modelo de consistência dos dados de disco, DSMIO se diferencia dos sistemas de arquivos distribuídos porque adota o modelo *Lazy Release Consistency* [1], enquanto que a quase totalidade dos outros sistemas adota o modelo seqüencial de consistência [73, 66] e os mais recentes adotam outros modelos relaxados [74][71].

O estudo a seguir visa avaliar a proposta do sistema DSMIO em adotar um modelo de consistência LRC ao invés de outros modelos relaxados, conforme mencionado acima. Para isto, desenvolvemos um sistema GOA+RC cuja cache compartilhada mantêm a coerência dos dados segundo o modelo de consistência por liberação para comparar o seu desempenho com o sistema GOA+DSMIO que adota o modelo de consistência por liberação preguiçosa [90].

5.2.1 Metodologia

Neste experimento nós substituimos o sistema GOA+NFS pelo sistema GOA+RC. Nesse sistema, cada processador teve a sua cache local substituída por uma cache

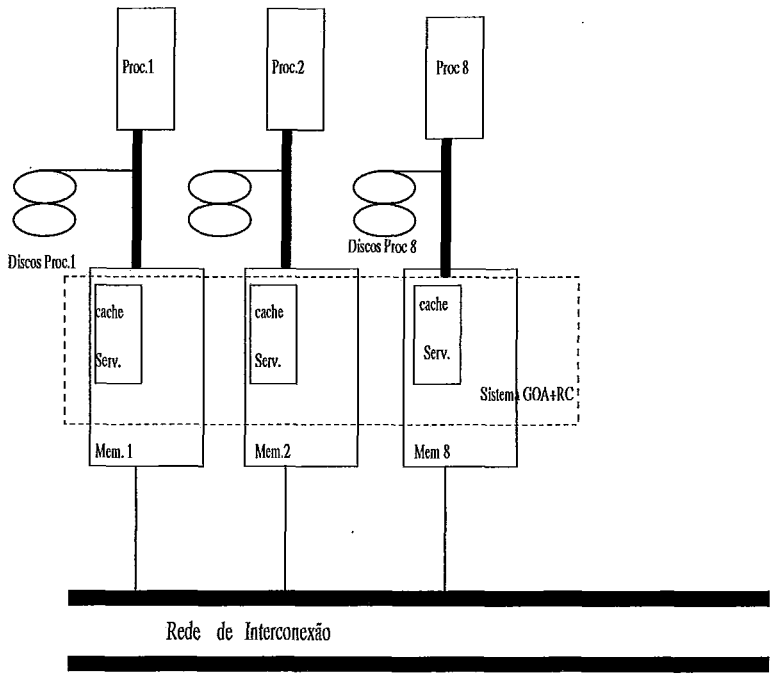

Figura 5.5: Sistema SGBD paralelo GOA acessando a base de dados via Sistema RC

compartilhada (figura 5.5, e cada processador acessa o seu disco local de forma similar ao sistema GOA+DSMIO, de forma que não é mais necessário acessarmos o disco via NFS. Neste teste nós utilizamos o *switch* do SP de 40MB/segundo. Nós mantivemos o grau de compartilhamento entre os objetos constante, ou seja, 100% dos objetos Atomic Parts modificados são compartilhados, e variamos o número de processadores de 1 a 8.

A base de dados é fragmentada pelos processadores do sistema de forma circular (*round-robin*). A fragmentação é tal que 50% dos objetos acessados por um processador são locais e 50% são remotos. Cada processador recebe uma faixa de objetos *Assembly Part* para ser percorrido pela *traversal* T3 de forma a atualizar todos os objetos *Atomic Part* encontrados.

O sistema GOA+RC é similar ao sistema GOA+DSMIO para operações de leitura. MAS para operações de escrita ele aplica um protocolo que obedece o modelo de consistência RC:

- Uma operação de leitura a um dado que não está presente na *cache* do nó cliente (cache GOA local) gera uma operação de pedido de leitura da página correspondente ao dado para o seu respectivo nó servidor (cache do servidor).
- Uma operação de escrita faz com que o sistema GOA+RC, após adquirir a

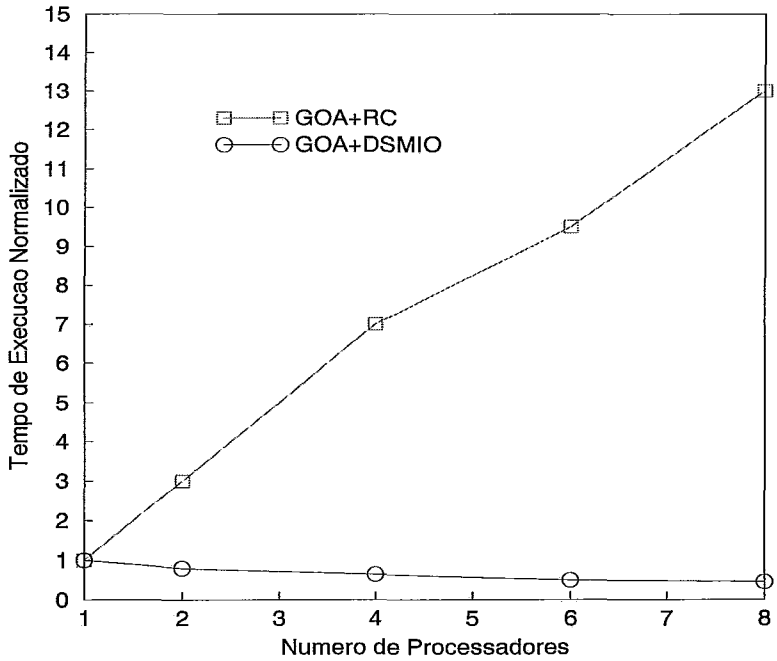

Figura 5.6: Tempo de execução do GOA+RC e do GOA+DSMIO em função do número de processadores.

operação de acquisição de um *lock* e após a operação de atualização do dado, envie para o servidor uma mensagem contendo a página modificada e a seguir envie mensagens de invalidação para os nós clientes que possuem cópia da página modificada, antes da operação de liberação do *lock*.

5.2.2 Análise de Resultados

Como no experimento anterior, nos concentramos apenas nos aspectos de coerência e consistência de memória dos sistemas e, além disso, nos ganhos gerados pelos mecanismos de *diffs*.

O nosso objetivo está em apresentar que os ganhos gerados pelo modelo LRC são devidos à redução na quantidade de mensagens e que os ganhos gerados pelos mecanismo de *diffing* são devidos à redução no tamanho das mensagens, e no espaço da área de armazenamendo de dados da memória local, devido ao modelo de consistência LRC e aos mecanismos de *diffing*.

Na figura 5.6, apresentamos as curvas de tempo de execução de **T3** em GOA+RC e GOA+DSMIO em função do número de processadores. Os tempos estão normalizados com relação ao tempo de execução de GOA+RC em 1 processador.

Observamos na figura que GOA+DSMIO apresenta ganhos crescentes de de-

sempenho em relação ao GOA+RC a medida em que aumentamos o número de processadores do sistema. Isso se deve a grande redução obtida por GOA+DSMIO no tempo de recuperação de um dado compartilhado e no tempo de propagação das modificações realizadas no dado compartilhado.

A redução obtida no tempo de recuperação do dado compartilhado se deve ao fato de que, nessa recuperação, GOA+DSMIO precisa somente receber e aplicar os *diffs* correspondentes a este dado, enquanto que GOA+RC necessita da última versão do dado executada pela operação de escrita no disco. Por exemplo, o tempo de recuperação de um dado compartilhado em GOA+DSMIO é de aproximadamente 5% do tempo de recuperação de um dado compartilhado em GOA+RC para 8 processadores.

A redução obtida no tempo de propagação das modificações realizadas no dado compartilhado é devida principalmente à diferença dos modelos de consistência de memória adotados pelos dois sistemas. GOA+RC emprega o modelo de consistência por liberação agressivo RC, que propaga as modificações do dado compartilhado para os outros processadores na saída da seção crítica. Mais especificamente, GOA+RC envia uma mensagem de atualização do dado para o servidor do mesmo e mensagens de invalidação para os outros processadores do sistema que compartilham o mesmo dado. O processador que escreveu o dado deve esperar *acknowledgements* de todos os processadores que receberam invalidações e um *acknowledgement* do servidor.

Já GOA+DSMIO emprega o modelo de consistência por liberação preguiçosa que diminui drasticamente o número de mensagens e a quantidade de dados trocados, atrasando a propagação das informações de coerência até o momento do próximo pedido de *lock*. Nesse momento, somente o processador que requisitou o *lock* recebe informações de coerência sobre os dados compartilhados. Dessa forma, GOA+DSMIO não gera mensagens de invalidação para cada nó cliente que possui cópia da página modificada assim como não necessita esperar pelas mensagens de *acknowledgements* e também executar o envio da escrita da página ao nó servidor. Na verdade, a saída de uma seção crítica em GOA+DSMIO envolve apenas a criação de *diffs* e seu armazenamento em memória principal, não gerando, portanto, nenhum tipo de comunicação pela rede. Comparando o sistema GOA+RC com o sistema GOA+DSMIO para 8 processadores, por exemplo, vemos que o número de mensagens transmitidas no GOA+RC é em média 86% maior que no sistema GOA+DSMIO.

Além disso, GOA+DSMIO também reduz o tamanho das mensagens transmitidas já que atualiza os dados através dos mecanismos de *diffs*. Por exemplo, para a aplicação *T3* o tamanho das mensagens de atualização dos dados são reduzidos em média mais de 99%.

Na figura, observamos também que GOA+DSMIO mostrou que a redução no tempo de execução não é proporcional ao aumento do número de processadores, o que se deve ao uso de uma base de dados pequena em nossos experimentos. GOA+RC, entretanto, apresenta aumento considerável do tempo de execução, *slowdown*, com o aumento do número de processadores. Nesse caso, o problema não é o tamanho da base mas o alto custo da coerência de dados em GOA+RC, aproximadamente 99% do seu tempo total de execução garantindo é devido as operações geradas pelo protocolo de escrita, para manter a coerência dos dados no disco antes de liberar cada *lock*, seja escrevendo os dados no disco no caso do servidor, seja enviando os dados atualizados para os seus respectivos servidores.

Testes com DSMIO adaptativo

No experimento discutido acima existe grande compartilhamento de dados. Entretanto, numa situação em que não há compartilhamento de dados na aplicação, o uso de *diffs* para armazenamento das modificações realizadas levaria o sistema DSMIO a gerar operações desnecessários de criação de *diffs* conforme vimos no teste da seção anterior. O que tornaria, certamente, mais lento que GOA+RC. Para evitar esse problema, o sistema DSMIO possui um mecanismo que permite a adaptação dinâmica em relação ao padrão de compartilhamento de dados gerados pela aplicação, conforme descrito na seção anterior. De forma a avaliar esta otimização, estudamos o desempenho de GOA+DSMIO para aplicações onde não há compartilhamento de dados e observamos que GOA+DSMIO envolve gastos desprezíveis nesses casos.

5.2.3 Conclusões

Os resultados apresentados na seção 5.1 comprovam a eficiência do sistema DSMIO de trazer a coerência dos dados armazenados em disco para o nível da memória principal. O estudo da seção 5.2 comprova a eficiência mostrada no contexto de sistema de *software DSM* em adotar um modelo de consistência LRC ao invés de modelos menos relaxados para a manutenção da coerência dos dados compartilhados

e de empregar mecanismos de *diffs* para o armazenamento dos dados escritos no disco.

Na verdade, nossos resultados mostram que os gastos gerados pela criação e posterior adaptação dos *diffs* são suplantados pelos ganhos que esse sistema obtém através da eliminação de acessos à disco.

Baseados nesses resultados preliminares iremos examinar nos próximos capítulos o aspecto de coerência de cache do sistema DSMIO no projeto de sistemas SGBDOO.

Capítulo 6

Proposta de um Sistema Cliente-Servidor DSMIO

Para avaliar o sistema DSMIO no universo das aplicações de SGBD, realizamos um estudo para determinar como as questões relativas à *cache* compartilhada são tratadas na área de Banco de Dados.

Observamos que os sistemas SGBD cliente-servidor possuem questões e soluções bastante próximas aos estudos desenvolvidos na área de *Software DSM* [8], e que diversos estudos relativos a questões de coerência de cache têm sido desenvolvidos nesta área [11, 35, 12, 10]. Desta forma, decidimos que, ao basearmos o estudo de avaliação de desempenho do sistema DSMIO seguindo o padrão dos estudos acima citados, estaremos adotando uma metodologia segura para obtermos resultados relevantes e significativos com relação ao seu desempenho.

De acordo com este padrão, comparamos o desempenho do sistema DSMIO em um ambiente cliente-servidor com relação ao estado da arte dos algoritmos de consistência de cache cliente-servidor, no caso o algoritmo de consistência de *cache Callback Locking* (CBL) [8]. Este estudo é apresentado no próximo capítulo.

6.1 Conceitos básicos sobre o SGBDOO cliente-servidor

O desempenho dos SGBDOO em aplicações que envolvam a execução de consultas complexas que recuperam uma grande quantidade de objetos e seus relacionamentos é ainda insatisfatório.

Uma forma de se aumentar o desempenho dos SGBDOO é adotar uma arquitetura cliente-servidor que divide os componentes de um sistema entre dois processos,

permitindo a distribuição de trabalho entre os processadores do sistema. Desta forma, por exemplo, enquanto um cliente analisa os resultados de uma consulta em sua estação de trabalho, o servidor estará disponível para o atendimento as solicitações de outros clientes [15].

A comunicação entre os processos cliente e servidor é geralmente realizada através de objetos ou páginas. Nos sistemas denominados servidores de objetos, os clientes requisitam objetos ao servidor que os recupera do banco de dados e os envia para os clientes. Nos sistemas denominados de servidores de páginas, a unidade de transferência entre o cliente e o servidor é um bloco físico de dados como páginas ou segmentos ao invés de objetos.

6.2 Consistência de cache em sistemas cliente-servidor

As questões relacionadas com a consistência de *cache* ocorrem em diversos tipos de sistemas distribuídos e/ou paralelos, incluindo sistemas de Banco de Dados de disco compartilhado assim como em sistemas que não são voltados para sistemas de Banco de Dados tais como sistemas multiprocessadores, sistemas de memória compartilhada distribuída, e sistemas de arquivos paralelos. Enquanto existem muitas similaridades entre as técnicas básicas de manutenção de consistência dos sistemas de Banco de Dados cliente-servidor e dos outros ambientes, existem diferenças significativas que definem as alternativas de projeto e as relações de custo-benefício. A seguir nós iremos descrever como a consistência de *cache* transacional difere da consistência de *cache* em ambientes que não são de Banco de Dados [8]. As principais diferenças são relacionadas com as seguintes questões:

- Critério de correção
- Granularidade de *caching*
- Custos de transmissão de mensagens
- Características de carga

6.2.1 Coerência de caches em Sistemas Multiprocessadores

Os primeiros trabalhos sobre coerência de cache, foram desenvolvidos dentro do contexto de multiprocessadores de memória compartilhada, de forma que um grande

número de protocolos de coerência foram desenvolvidos e estudados para tais sistemas [24]. A noção tradicional de correção em sistemas multiprocessadores é chamada de consistência seqüencial [26] e tem como objetivo assegurar que a execução do programa em um ambiente multiprocessador de memória distribuída forneça resultados similares à execução do mesmo programa em um sistema uniprocessador de memória única executanto de forma *multi-threaded.* Este modelo de consistência se preocupa com a ordenação individual dos acessos a memória e não atende as questões relacionadas com as unidades de acesso das transações necessárias em um sistema de banco de dados.

Trabalhos mais recentes na área de multiprocessadores desenvolveram modelos alternativos para melhorar o desempenho do sistema necessitando como contrapartida que o programador e/ou compilador venha a inserir explicitamente primitivas de sincronização aos códigos a serem paralelizados [25]. Muitos destes modelos, tal como o *Release consistency* [42], permitem que os programadores possam combinar múltiplos acessos a memória em unidades protegidas pelas variáveis de sincronização. Entretanto, para garantir a corretude do programa, é necessário que a composição das unidades de sincronização sejam conhecidas e aceitas *a priori* por todos os processos cujos acessos possam afetar as localizações da memória. Este conhecimento *a priori* é um requisito razoável em tais ambientes, já que o objetivo principal é o de fornecer uma corretude de execução similar a um sistema com um programa único executanto de forma *multi-threaded.* O compartilhamento dos dados entre programas distintos não é diretamente atendido por tais protocolos. Além disso, todos esses modelos têm como objetivo fornecer semânticas que se aproximam da memória volátil de um sistema uniprocessador, de forma que eles também não fornecem suporte para a tolerância a falhas.

Sistemas de Banco de Dados necessitam de uma série de requisitos que não são oferecidos pelos algoritmos de memória de sistemas multiprocessadores. As semânticas de atomicidade, consistência, isolamento e durabilidade, comumente chamadas de ACID, são necessárias para fornecer corretude para a execução concorrente de transações contendo grupos arbitrários de operações que asseguram uma execução correta mesmo na presença de falhas. Isto é importante devido ao fato de que sistemas de Banco de Dados têm que fornecer suporte a acessos corretos a banco de dados compartilhados na presença de uma carga de trabalho que é constantemente alterada. As cargas de trabalho das bases de dados são tipicamente sujeitas a

diferentes usuários, que acessam o sistema de forma concorrente com uma mistura de operações de consultas *ad hoc* e de operações de atualizações. Como resultado das diferenças de foco entre os dois sistemas, as técnicas básicas que tem sido desenvolvidas para sistemas de Banco de Dados, tal como o algoritmo de *two-phase locking* ou o algoritmo *optmistic concurrency control* [10], não são empregados em ambientes de multiprocessadores, e são componentes fundamentais para os sistemas de Banco de Dados cliente-servidor.

Apesar das diferenças de enfoque entre os dois sistemas, existem questões básicas comuns entre os sistemas que são abordadas por todos os algoritmos de coerência de *cache*. Por exemplo, cópias que são desatualizadas nas *caches* podem ser tratadas de duas maneiras; seja através de invalidações, seja através da propagação do dado atualizado i.e. *write-invalidate and write-broadcast* [24]. Da mesma forma, as ações de coerência podem ser distribuídas através da utilização de um meio de *broadcast*, ou seja, *snooping caches* [27], ou pode ser mantido por diretórios [28]. Mesmo neste nível, as diferenças de arquitetura entre um sistema de Banco de Dados de ambiente *data-shiping* e de um sistema multiprocessador, tais como custo das mensagens, granularidade da cache, granularidade de acesso a *cache* limitam que um sistema possa ser adotado pelo outro.

6.2.2 Consistência em sistemas de *software* DSM

Sistemas de *Software* de Memória Distribuída (DSM) possuem uma relação de custo-benefício próxima aos sistemas de Banco de Dados de ambientes *data-shipping* [37], que são sistemas Gerentes de Banco de Dados que fornecem os dados para serem tratados nos nós clientes . Sistemas DSM fornecem a abstração de um espaço de endereçamento de uma memória compartilhada virtual única, que atinge todos os nós do sistema distribuído. De forma diferente do sistema de *cache* de um sistema multiprocessador, que cujo suporte é desenvolvido pelo sistema de *hardware*, os sistemas de *software* DSMs tipicamente necessitam de uma assistência mínima de *hardware*, fornecida pelos sistemas de acesso às páginas de memória virtuais dos sistemas operacionais modernos. Com relação a granularidade de acesso a *cache*, os sistemas DSMs são similares ao sistemas de Banco de Dados cliente-servidor, devido ao fato de que a unidade de consistência é composta seja por páginas seja por um grande número de linhas de *cache*. Em termos de relação de custo-benefício, os *software* DSMs também estão mais próximos dos sistemas servidores de páginas do que

os sistemas multiprocessadores de memória compartilhada, devido ao fato de que a manutenção da coerência da cache é mantida através de mensagens. Ou seja, apesar de um sistema de *software* DSM ser construído em um sistema multiprocessador, as mensagens transmitidas são mais eficientes. As principais diferenças com relação ao sistema cliente-servidor estão relacionadas com os critérios de corretude dos sistemas DSM, que são tipicamente os mesmos para os sistemas multiprocessadores.

6.2.3 Trabalhos Relacionados com sistemas DSM/LRC

Devido ao alto custo das mensagens nos ambientes distribuídos, melhorias recentes nos protocolos de coerência tal como as adotadas pelo modelo *Lazy Release Consistency* [1], exploram os padrões de comunicação entre os processadores para reduzir ainda mais os gastos com as mensagens no sistema. Alguns protocolos para sistemas de Banco de Dados em ambiente *data-shipping* também tentam explorar os padrões de comunicação existentes em ambiente de banco de dados, entretanto, estes padrões são determinados em grande parte pela capacidade dos protocolos em atender os critérios das transações ACID. De novo, as principais diferenças com relação ao sistema cliente-servidor estão relacionadas com os critérios de corretude, que são tipicamente os mesmos para os sistemas multiprocessadores.

6.3 Algoritmos de coerência de cache em ambiente cliente-servidor

Em um ambiente SGBD cliente-servidor, quando dois ou mais nós clientes executam operações de atualização em dados comuns, diversas cópias destes dados encontram-se armazenadas em diferentes *caches* dos nós do sistema. Desta forma, se torna necessário assegurar que cada uma das modificações não só seja transmitida para todos as respectivas cópias no sistema como também assegurar que as atualizações sejam executadas de forma seqüencial. Definimos um dado como estando em estado inconsistente na *cache* quando o dado não foi atualizado após operações de validações, executadas pelos outros nós do sistema, chamadas de *commit*. Para evitar que isto aconteça, diversos protocolos de consistência de *cache* para sistemas cliente-servidor foram objeto de estudos nos últimos anos e dezenas de algoritmos foram propostos, entre eles se destacam [30] [10] [31].

Podemos classificar esses algoritmos em duas classes, a Preventiva e a Corretiva

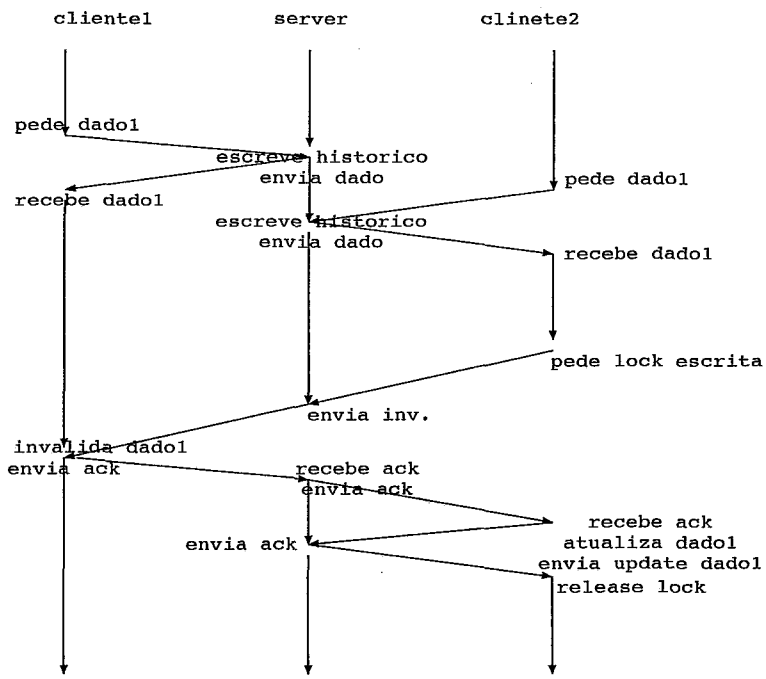

Figura 6.1: O algoritimo CBL.

[8]. Algoritmos do tipo preventivo impedem o acesso a dados inconsistentes durante uma transação, enquanto que os algoritmos do tipo corretivo permitem o acesso ao dado inconsistente mas detectam e executam operações para manter a consistência do dado no momento da operação de *commit*. A maioria dos sistemas comerciais adotam algoritmos do tipo preventivo devido à existência de aplicações que não podem tolerar as altas taxas de interrupcão dos algoritmos do tipo corretivo.

De acordo com [7], [8], [12] [11], [9], o algoritmo do tipo preventivo que apresenta melhor desempenho é o chamado *Call-back Locking* (CBL) [10], [11], [32] e [33] e é o algoritmo mais amplamente adotado em SGBDOO. Isto se deve ao fato de que este algoritmo une características de bom desempenho e de baixa taxa de abortos em transações [11]. Devido a esses motivos, escolhemos o algoritmo CBL como referência para avaliarmos o sistema DSMIO em um ambiente de Banco de Dados cliente-servidor.

6.3.1 O algoritmo CBL

No algoritmo CBL [8], como pode ser visto pela figura 6.1, o cliente inicialmente envia o pedido de leitura de um dado ao servidor. O servidor, após receber o pedido de leitura, envia ao cliente uma cópia da página relativa ao dado em conjunto com o seu respectivo *lock* (implícito) de leitura. Uma vez que a página e o *lock* chegam

ao cliente, eles poderão ser mantidos na *cache* do cliente mesmo após o término da transação. (Ao receber uma página, o cliente recebe de fato do servidor um *lock* de leitura implícito da página, isto porque efetivamente, o cliente recebe apenas o dado do servidor.) Do ponto de vista do servidor, o cliente possui o *lock* de leitura enquanto ele mantiver a página na sua *cache*. Já os *locks* de escrita são pedidos explícitos ao servidor e são devolvidos ao término da transação. Quando um cliente pede um *lock* de escrita que entra em conflito com um ou mais *locks* de leitura concorrentemente armazenados nas *caches* de outros clientes, o servidor faz um pedido de *call back* para os clientes que possuem os *locks* de leitura. Quando um cliente recebe um pedido de *call back* do servidor, ele verifica se a página está sendo acessada. Caso a página não esteja sendo acessada, o cliente remove a página da sua *cache* e envia uma mensagem de *acknowledgement* para o servidor. Caso a página esteja sendo usada, o cliente coloca o pedido em uma fila e envia uma mensagem para informar ao servidor que a página está sendo usada.

Esta mensagem permite que o servidor possa fazer uma deteção de *deadlock*. O servidor irá enviar o *lock* de escrita apenas depois que todos os *locks* de leitura tenham sido liberados pelos outros clientes. Uma vez enviado o *lock* de escrita para o cliente que solicitou o *lock*, qualquer pedido subseqüente tanto de *lock* de escrita quanto de *lock* de leitura para a página será retido no servidor até que o *lock* de escrita seja liberado pelo cliente que está com ele. No final da transação o cliente que possui o *lock* de escrita envia uma cópia da nova página para o servidor e libera o *lock*, mantendo uma cópia da página na sua *cache*.

Devido ao fato de que um cliente recebe do servidor páginas e seus respectivos bloqueios de leitura implícitos para serem armazenados na sua *cache*, o servidor necessita ser informado pelo cliente quando as páginas vierem a ser removidas da sua *cache*. Para manter o servidor informado das remoções das páginas da sua *cache*, o cliente acrescenta uma informação contendo os números das páginas que foram removidas da sua *cache* na próxima mensagem a ser enviada ao servidor. Desta forma a tabela do servidor apresenta um panorama atualizado das localizações das páginas cacheadas pelo sistema.

Modelo de Consistência

O algoritmo CBL apresenta um modelo de consistência similar ao modelo de consistência RC [42]. Em CBL, antes que um *lock* de escrita possa ser liberado, todas

as escritas relativas à região crítica protegida pelo *lock* têm que ter sido atualizadas nos seus respectivos servidores e todas as cópias relativas presentes em outros nós clientes têm que ter sido invalidadas.

No modelo RC (seção 2.2.4), a idéia básica é que se os pontos de sincronização podem ser identificados, então somente nestes pontos a memória precisa estar consistente. O modelo de consistência por liberação requer apenas que as operações tenham terminado antes que uma liberação de uma região crítica seja efetuada.

A seguir faremos uma descrição da versão dos sistemas desenvolvidos para avaliar o desempenho do algoritmo de coerência de cache do sistema DSMIO com relação ao algoritmo CBL. Os sistemas desenvolvidos são respectivamente o sistema GOA-DSMIO E GOA-CBL. Iniciaremos com a descrição do sistema GOA-CBL.

6.4 O sistema cliente-servidor GOA-CBL

Desenvolvemos as primitivas de sincronização de aquisição de *lock* e de liberação de *locks* de leitura e de escrita de dados, de acordo com o protocolo determinado pelo algoritmo CBL. O sistema foi desenvolvido em um ambiente SP/IBM real, utilizando 17 processadores, 1 nó servidor e 16 nós clientes. O avaliação do desempenho é realizada com o número de processadores fixos, no caso 17 processadores, variando a carga das aplicações, segundo medotologia [8] adotada para se avaliar o desempenho de algoritmos de coerência de cache em sistemas cliente-servidor.

A figura 6.2 apresenta um sistema cliente-servidor SGBDOO GOA-CBL onde o processador **P1** é o nó servidor que atende os pedidos de páginas de 16 nós clientes. Cada nó cliente possui uma cache local, que é a *cache* local do processo gerente de banco de dados,GOA, de forma que uma mesma página pode possuir cópia em diversas caches de processadores clientes do sistema.

Um nó **pai**, no caso o nó bf P1 inicializa um processo GOA em cada um dos nós clientes através das rotinas de inicialização do *Treadmarks*. Neste teste, este nó irá executar apenas o processo servidor de dados do disco.

Cada um dos 16 nós cliente executa um processo Gerente de de Banco de dados, GOA, que ir'a atender aos pedidos das consultas do processo de navegação. O processo gerente irá solicitar as páginas dos dados a serem atendidos pelos pedidos de consulta ao nó servidor.

Um processo de navegação pela base poderá acessar dados compartilhados por

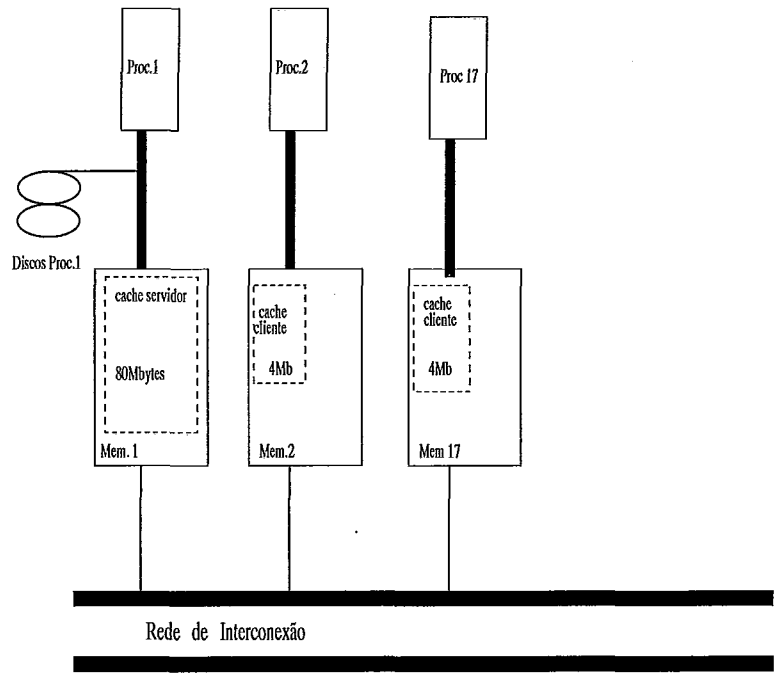

Figura 6.2: O sistema SGBDOO cliente-servidor GOA-CBL.

outros processadores. Para manter a coerência dos dados, o pedido necessita de ser protegido por pedidos de aquisição e de liberação de *locks* de leitura e de escrita das páginas.

O nó servidor armazena as páginas lidas do disco em uma cache de 80 Mbytes, que é uma cache de tamanho grande o suficiente para que não ocorra necessidade de execução de operações de *flush* parcial. No momento nós estamos interessados em avaliar o desempenho do algoritmos sem a interferência do sistema de *flush* parcial das caches. As páginas são enviadas para os processos cliente, no caso os processos GOA que armazenam as páginas em uma *cache* local 4Mbytes (tamanho da cache original de um processo GOA, que não foi alterado)

O nosso sistema trabalha com pedidos de bloqueio ao nível da página. Desta forma um nó cliente, ao solicitar um pedido de bloqueio ou liberação de um dado, tem que solicitar ao nó servidor o bloqueio ou liberação da página na qual o dado está inserido.

As páginas possuem 2kbytes, que é o mesmo tamanho que os blocos lidos do disco, assim como o mesmo tamanho das páginas armazenadas nas caches dos nós clientes e do nó servidor. O tamanho de 2kbytes está relacionado com os parâmetros do processo GOA.

O algoritmo implementado obedece o protocolo apresentado na seção anterior,

ilustrado pela figura 6.1, ou seja:

Quando um nó cliente deseja ler um dado, que não está armazenado na sua *cache* local, ele envia um pedido de *lock* de leitura da sua respectiva página ao nó servidor. Uma vez recebido o dado, ele é armazenado na sua *cache local* e a execução da aplicação prossegue normalmente. Ao receber o dado do nó servidor, o nó cliente recebe um *lock* de leitura implícito da página.

Um nó servidor, ao receber um pedido de *lock* de leitura de uma página, caso ele não tenha esta página armazenada na sua *cache* local, irá ler do disco, armazenar na sua *cache* local e enviar ao nó cliente. O servidor grava a informação em de que o dado foi enviado para o nó cliente em um arquivo. Desta forma o servidor irá saber quais são os nós clientes que possuem cópias daquele dado.

Um nó cliente, ao executar um pedido de *lock* de escrita, irá enviar o pedido para o servidor e após receber a mensagem de autorização, irá enviar a página atualizada para o servidor e enviar em seguida um pedido de liberação do *lock*.

Um nó servidor, ao receber um pedido de *lock* de escrita de uma página, irá consultar a sua tabela de informação para saber se a página está sendo bloqueada por um outro nó cliente. Neste caso ele enviará uma mensagem de negação de *lock* para o nó cliente.

Caso contrário, ele irá consultar a sua tabela para saber quais são os nós clientes que possuem cópias da página e enviar um pedido de invalidação para estes nós e após receber a mensagem de autorização de invalidação destes nós, irá enviar a mensagem de autorização de *lock* de escrita. Antes de enviar a autorização do *lock* de escrita, o nó servidor irá atualizar o estado das páginas na sua tabela de informação.

Um nó servidor, ao receber uma mensagem de atualização de uma página de um nó cliente, irá escrever a página na sua *cache*.

Um nó servidor ao receber um pedido de liberação e uma página de um nó cliente irá liberar o *lock*.

Ao término da execução de todas as aplicações dos nós clientes, todos os nós do sistemas executam uma operação de barreira do *TreadMarks* para que o servidor possa ter a informação de que não ocorrerá mais pedidos dos nós clientes. Após a execução da operação de barreira, o nó servidor irá executar uma operação para escrever os dados atualizados de volta no disco.

Por fim, os nós do sistema executam a operação *Tmk-exit* para que os processos do sistema sejam finalizados e seus respectivos arquivos de saída sejam enviados

para o nó servidor.

Desenvolvemos as seguintes rotinas para implementarmos o sistema GOA-CBL:

- Pedido de *lock* de leitura
- Pedido de *lock* de escrita
- Envio de *lock* de leitura
- Envio de *lock* de escrita
- Envio de atualização de página para o servidor
- Pedido de liberação de *lock* de leitura
- Pedido de liberação de *lock* de escrita
- *Flush* da cache do servidor para o disco.

Em resumo, utilizamos o sistema de *software DSM Treadmarks* para gerar uma versão paralela do sistema GOA. Neste teste um processo pai **P1** inicializa, através das rotinas de paralelização do *Treadmarks*, um processo GOA em cada um dos nós clientes, assim como as aplicações *traversal* a serem executadas em cada um dos nós clientes. Ao término da execução das aplicações o processo pai finaliza os processos dos nós clientes.

No processador **P1** nos desenvolvemos um processo servidor de dados, para atender os pedidos dos nós clientes. Para o desenvolvimento deste servidor, nós implementamos as rotinas acima citadas.

6.5 O sistema cliente-servidor GOA-DSMIO

A figura 6.3 apresenta um sistema cliente-servidor SGBDOO GOA-DSMIO onde o processador P1 é o nó servidor que atende os pedidos de páginas de 16 nós clientes. Cada nó cliente possui uma cache local, de forma que uma mesma página pode possuir cópia em diversas caches de processadores clientes do sistema. As caches locais dos processadores clientes foram substituídas pela cache compartilhada do sistema DSMIO.

No sistema GOA-DSMIO paralelizamos o sistema GOA uniprocessador utilizando o sistema *TreadMarks* para criar um processo cliente GOA em cada um dos 16

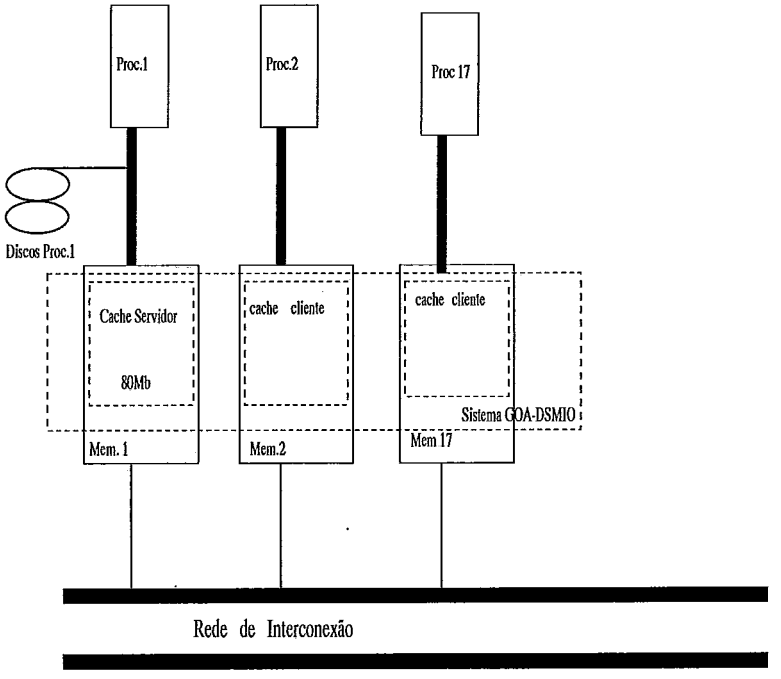

Figura 6.3: O sistema SGBDOO cliente-servidor GOA-DSMIO.

nós processadores clientes e um processo servidor no nó de inicialização do sistema. Utilizamos as primitivas de sincronização de aquisição e de liberação de páginas do sistema *Treadmarks*.

O nó servidor inicializa um processo GOA em cada um dos nós clientes através das rotinas de inicialização do *Treadmarks*.

No sistema GOA-DSMIO, cada nó cliente executa um processo GOA uniprocessador contendo uma *cache compartilhada* de 80Mbytes e utiliza as rotinas de pedido de aquisição e liberação do sistema *TreadMarks*. Ele executa um processo que envia pedidos de leitura de páginas ao nó servidor para serem armazenadas na *cache* compartilhada e que gera *diffs* das páginas modificadas quando ocorre um pedido de escrita em uma página pertencente à memória compartilhada.

O nó servidor executa um processo que atende pedidos de leitura e escrita de páginas dos nós clientes. As páginas são lidas do disco e armazenadas em uma *cache compartilhada* de 80 Mbytes.

As páginas possuem 2kbytes, que é o mesmo tamanho que os blocos lidos do disco, assim como o mesmo tamanho das páginas armazenadas nas caches dos nós clientes e do nó servidor.

A figura 6.4 apresenta o algoritmo do sistema SGBDOO cliente-servidor GOA-DSMIO.

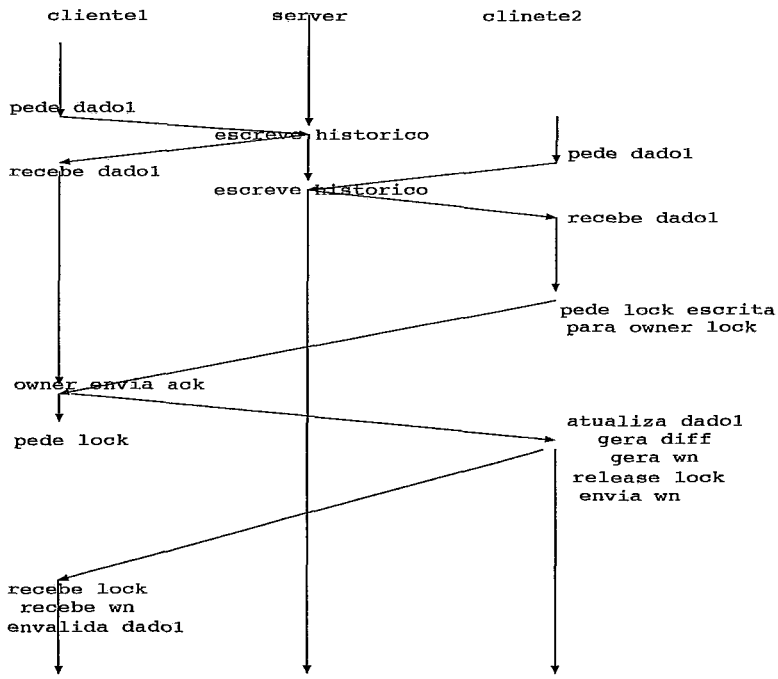

Figura 6.4: O algoritmo GOA-DSMIO.

Quando um nó cliente deseja ler um dado que não está armazenado na sua *cache compartilhada*, ele envia um pedido de leitura da sua respectiva página ao nó servidor. Uma vez recebido o dado, ele é armazenado na *cache compartilhada* e a execução da aplicação prossegue normalmente.

Um nó servidor, ao receber um pedido de de leitura de uma página, caso ele não tenha esta página armazenada na *cache* compartilhada, irá ler do disco, armazenar na sua *cache* compartilhada e enviar ao nó cliente.

Um nó cliente, ao executar um pedido de *lock* de escrita, irá enviar o pedido para o nó servidor do *lock* após receber a mensagem de autorização do último nó que liberou o *lock*, irá gerar um *diff* da página atualizada e a seguir enviar um pedido de *lock*. O nó servidor do *lock*, possui a informação sobre o nó que solicitou o pedido de *lock* mais recentemente e transmite esta informação para o nó que está pedindo o *lock*, que por sua vez envia o pedido para o nó que possui o *lock* da página. Uma vez liberado o *lock*, este último nó irá transferir para o nó pedinte todas as informações relacionadas com os intervalos entre o *vector timestamp* dos dois nós, de acordo com o protocolo *Treadmarks*.

Assim como no GOA-CBL, ao término da execução de todas as aplicações dos nós clientes, todos os nós do sistema executam uma operação de barreira do *TreadMarks* para que o servidor possa ter a informação de que não ocorrerá mais pedidos dos

nós clientes. Após a execução da operação de barreira, o nó servidor irá executar uma operação para escrever os dados atualizados de volta no disco. Entretanto no caso do sistema GOA-DSMIO, é executada uma operação *Tmk-Flush*, que é uma operação que envolve a recuperação e aplicação de todos os *diffs* do sistema, para então escrever a página atualizada de volta ao disco, conforme pode ser visto em detalhes no capítulo sobre a descrição do sistema DSMIO.

Finalmente, os nós do sistema executam uma operação de *Tmk-exit* para que os processos do sistema sejam finalizados e seus respectivos arquivos de saída sejam enviados para o nó servidor.

De forma diferente do sistema GOA-CBL, este sistema utiliza as seguintes rotinas do sistema DSMIO:

- Tmk-read-opt
- Tmk-write-opt
- Tmk-Flush

E as rotinas de sincronização de bloqueio do *TreadMarks: Tmk-lock-acquire e Tmk-lock-release.*

Em resumo, o sistema GOA+DSMIO é um sistema DSMIO desenvolvido para 17 processadores onde o processador pai, o processador P1, é o único nó *home* do sistema, e fica dedicado a atender os pedidos de dados dos 16 nós clientes do sistema.

Capítulo 7

Validação do Sistema SGBDOO Cliente-Servidor GOA+CBL

Nesse capítulo iremos avaliar comparativamente o desempenho do sistema SGBDOO cliente-servidor GOA+DSMIO com o do sistema cliente-servidor GOA+CBL para diferentes cargas de trabalho assim como diferentes configurações de sistema. Iremos adotar a metodologia de estudo de desempenho similar aos vários estudos recentes de desempenho de consistência de cache para sistema cliente-servidor [11] [12] [35].

7.1 Metodologia

Utilizamos o mesmo sistema IBM-SP. O nosso estudo adota uma arquitetura do tipo cliente-servidor com servidor de páginas, na qual a responsabilidade pelo gerenciamento dos dados do disco pertence ao servidor, descrito em detalhes no capítulo anterior.

Consideramos um cenário onde 100% do processamento é repartido entre os nós clientes devido ao fato de não possuirmos um sistema tradicional no qual existe um nó servidor com alta capacidade de processamento. Os nossos testes foram executados no sistema SP2 onde tanto o nó servidor quanto os nós clientes possuem a mesma capacidade. Desta forma o nó servidor é dedicado apenas à tarefa de servir as páginas requisitadas pelos nós clientes.

Nós não consideramos o cenário de *query shipping* ao servidor, já que neste cenário, a consulta é enviada pelo nó cliente para ser processada no nó servidor, de forma que levanta aspectos que não são de interesse em nosso estudo. Estamos interessados em estudar o controle de concorrência e de consistência de *cache*. Desta forma é necessário que o trabalho da aplicação seja executado em ambos os sistemas

cliente-servidor. Os estudos *query shipping* são importantes para avaliar apenas o trabalho sendo executado no servidor (tal como navegações de consultas).

7.1.1 Modelo da carga de trabalho

O *Benchmark 007* [4] foi desenvolvido para estudar o desempenho de SGBDOO. Entretanto este *Benchmark* não foi especificado de forma satisfatória para estudos sobre controle de concorrência [13]. Ele não inclui padrões de compartilhamento de dados nem valores sobre tamanhos de transações para se determinar o nível de contenção de dados do sistema. Como o nível de contenção do dado é um componente fundamental para qualquer estudo de consistência e de controle de concorrência de *cache*, nós iremos adotar, assim como em estudos anteriores [12] [11] características relevantes do *Benchmark 007*, tal como a noção de buscas *traversal*, o conceito de regiões privadas e compartilhadas, o padrão da base de dados média *(working sets)* e os tamanhos dos objetos atômicos, e acrescentar um padrão de compartilhamento de dados e tamanhos de transações próprios. Desta forma o padrão de compartilhamento de dados irá especificar o número de conflitos de leitura/escrita e de escrita/escrita. Neste estudo, assim como nos estudos anteriores [11] [12], nós examinaremos 3 padrões de compartilhamento; o padrão **Privado, o Compartilhado e o de Alta Contenção**, respectivamente [12] [11]. Desta forma estaremos cobrindo um espectro bastante amplo dos níveis de contenção de dados e ao mesmo tempo determinando a robustez do algoritmo de consistência de cache.

A carga de trabalho **privado** não apresenta contenção de dados. A contenção dos dados aumenta progressivamente com as cargas **compartilhado e alta contenção**, respectivamente. A base de dados consiste de uma série de regiões privadas (uma para cada cliente), de uma região compartilhada comum e de uma região de páginas chamadas de *left-over* que não são acessadas. Na carga de trabalho **privada** cada cliente acessa apenas os dados da região privada 80% tempo e acessa a região compartilhada 20% do tempo. Cada cliente atualiza os dados da sua região privada.

Na carga de trabalho **compartilhada** cada cliente acessa o dado da sua região privada 80% do tempo e da região compartilhada 10% do tempo e também outras regiões da base de dados, incluindo as regiões privadas dos outros clientes 10% do tempo. Os clientes podem atualizar objetos em todas as regiões.

Na carga de trabalho de **alta contenção**, cada cliente acessa dados da região compartilhada 80% do tempo e do resto da base de dados 20% do tempo. Os clientes

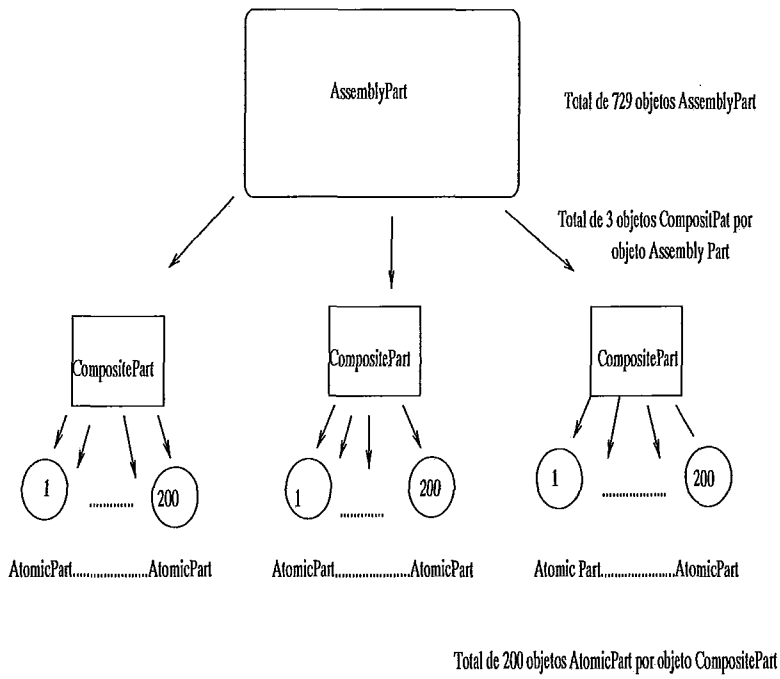

Figura 7.1: Navegação da aplicação T3

podem atualizar objetos de qualquer região.

Assim como no teste de desempenho adotado em [11], faremos um teste com um nó servidor e 16 nós clientes e iremos adotar os padrões de busca do *Benchmark 007* e as especificações da sua base média. Adotamos como padrão de navegação a *traversal* **T3** do *Benchmark 007* conforme mostra a figura 7.1, descrita com detalhes nesta seção para melhor explicar o padrão de medida de desempenho adotado.

Conforme podemos observar na figura 7.1, a *traversal* T3 navega pela hierarquia de *AssemblyPart/CompositePart/ AtomicPart* e percorre todos os objetos *AtomicPart* pertencentes a cada objeto *Composite Part.* Cada objeto *AtomicPart* percorrido será atualizado ou não segundo a porcentagem de objetos escritos, e o seu grau de compartilhamento do objeto escrito irá depender do padrão de carga adotado, ou seja, **private, compartilhado e alta contenção.**

De forma análoga a estudos anteriores [11], uma transação será formada pela navegação e busca por 200 objetos de 50kbytes.

Cada operação de uma transação pode acessar vários objetos pertencente a uma mesma página. A probabilidade de escrita determina quantos objetos serão atualizados em uma transação. Uma vez acessado um objeto o cliente pode executar uma operação de leitura ou de escrita. A probabilidade de escrita de um objeto irá determinar se será executada ou não uma operação de atualização.

7.1.2 Validação

Neste estudo, assim como nos estudos anteriores [11] [12], nós examinamos 3 padrões de compartilhamento de carga; o padrão **privado, o compartilhado e o de alta contenção** para comparar o desempenho dos sistemas cliente-servidor GOA-CBL,ver figura 6.2, e GOA-DSMIO, ver 6.3.

Os resultados obtidos mostraram que o sistema GOA-DSMIO apresenta um desempenho muito superior ao do sistema GOA-CBL para todas as aplicações estudadas, conforme apresentaremos em detalhes a seguir. Os sistemas foram comparados segundo o número de mensagens por transação em relação a percentagem dos objetos escritos, variando esta percentagem de 5% a 20% e comparados segundo o número de transações por segundo em relação a percentagem dos objetos escritos, variando esta percentagem de 5% a 20%.

Entretanto, para que estes resultados sejam relevantes, necessitamos primeiro avaliar se o sistema desenvolvido GOA-CBL é um sistema válido para o estudo de desempenho do sistema GOA-DSMIO.

Os estudos de comparação de desempenho de algoritmos de *cache* de sistemas cliente-servidor têm sido realizados através de simulação. Escolhemos o estudo [11] como base de comparação do sistema GOA-CBL, por ele ser o estudo mais recente em que os estudos de validação foram desenvolvidos. Na verdade as evoluções que a seguiram dizem respeito a unidade de transferência entre o sistema cliente-servidor.

Este estudo apresenta um algoritmo simulado *Adaptive Call Back Locking*, ACBL, que é um algoritmo *Call Back Locking* que adapta o pedido de bloqueio seja ao nível da página seja ao nível do objeto conforme descrito no trabalho [12]. A simulação deste trabalho, assim como trabalhos anteriores, foi executada em um simulador cujo ambiente de sistema possui as características apresentadas na tabela 7.1.

Podemos observar que existem várias características implementadas no sistema simulado que não tiveram possibilidade de serem replicadas no ambiente real.

A primeira diferença está nos SGBDOO utilizados. O estudo simulado se baseia em uma arquitetura SGBDOO cliente-servidor simulada segundo os parâmetros do trabalho [36], enquanto que o GOA-CBL se baseia em um SGBDOO real, o GOA [15].

A segunda está no fato de que o sistema cliente-servidor GOA-DSMIO é um sistema servidor de página, logo o sistema GOA-CBL desenvolvido também obedece a

Tabela 7.1: Parâmetros de sistema simulado [11]

Descrição	valor simulado
Vel.da CPU do cliente	50MIPS
Vel.da CPU do Servidor	150MIPS
Tamanho da cache do cliente	25% do BD
Tamanho da cache do Servidor	50% do BD
Discos do servidor	4 discos
Tempo de acesso de busca no disco	1600ucroseg/kbytes
Tempo de acesso de uma instr. no disco	1000uicrosecs/kbyte
Banda Passante da Rede	80Mips
Custo de setup do disco	5000 ciclos
Tamanho da base de dados	1300
Tamanho da Página	4k
Tamanho do Objeto	100 bytes
Número de clientes	16

mesma característica, ou seja, nós implementamos um CBL servidor de páginas para avaliar o desempenho do sistema GOA-DSMIO, e não um algoritmo CBL adaptativo para página ou objeto conforme o referido estudo.

Outras diferenças importantes estão apresentadas nas tabela 7.1 e na tabela 7.2 e são relacionadas com as diferenças entre as caracteristicas adotadas pelo sistema simulado e pelas características do sistema real utilizado para os testes, o sistema IBM-SP2.

Por fim ainda temos as diferenças relativas às aplicações. Embora use o *Benchmark 007*, o sistema simulado especifica poucos parâmetros relativos aos parâmetros da carga de trabalho. Mesmo assim, o sistema real tentou se aproximar o máximo possível desses parâmetros conforme pode ser observado pela mesma tabela, estando limitado pelas caracteristicas do *switch*, e pela falta de dados relativos à aplicação utilizada para a implementação das cargas de teste. Escolhemos então para implementarmos as cargas especificadas a *traversal T3* do *Benchmark 007*.

Abaixo iremos comparar os resultados gerados pelos sistemas ACBL-Simulado com o sistema GOA-CBL real para as cargas **privada, compartilhada e alta contenção.**

Carga Privada

Na figura 7.2 apresentamos o desempenho da aplicação **Privada** nos sistemas cliente-servidor GOA-CBL e ACBL-Simulado. O gráfico compara o número de transações

Tabela 7.2: Parâmetros de sistema real

Descrição	valor real
Vel.da CPU do cliente	65MIPS
Vel.da CPU do Servidor	65MIPS
Tamanho da cache do cliente	5% do BD
Tamanho da cache do Servidor	80% do BD
Discos do servidor	1 disco
Tempo de acesso de busca no disco	1600ucroseg/kbytes
Tempo de acesso de uma instr. no disco	1000ucrosecs/kbyte
Banda Passante da Rede	40Mbytes
Custo de setup do disco	5000 ciclos
Tamanho da base de dados	2600
Tamanho da Página	2k
Tamanho do Objeto	60 bytes
Número de clientes	16

por segundo em relação a porcentagem de dados escritos. Esta porcentagem aumenta de 5% a 20%.

Observamos na figura que à medida em que a percentagem de escrita aumenta, a taxa de transações por segundo de ambos os sistemas diminui.

Observamos também que a taxa de transações do sistema GOA-CBL é bem maior que o sistema ACBL-Simulado, e que a diferença diminui significativamente a medida em que se aumenta a percentagem de escrita do sistema.

Carga Compartilhada

Na figura 7.3 apresentamos o desempenho da aplicação **compartilhada** nos sistemas cliente-servidor GOA-CBL e ACBL-Simulado. O gráfico compara o número de transações por segundo em relação a porcentagem de dados escritos. Esta porcentagem aumenta de 5% a 20%.

Observamos na figura que a medida em que a percentagem de escrita aumenta, a taxa de transações por segundo de ambos os sistemas diminui.

Observamos também que a taxa de transações do sistema GOA-CBL é bem maior que o sistema ACBL-Simulado, e que a diferença se mantém significativa à medida em que se aumenta a percentagem de escrita do sistema. MAs não podemos tirar conclusões concretas desta figura sem traçar um paralelo entre as duas implementações.

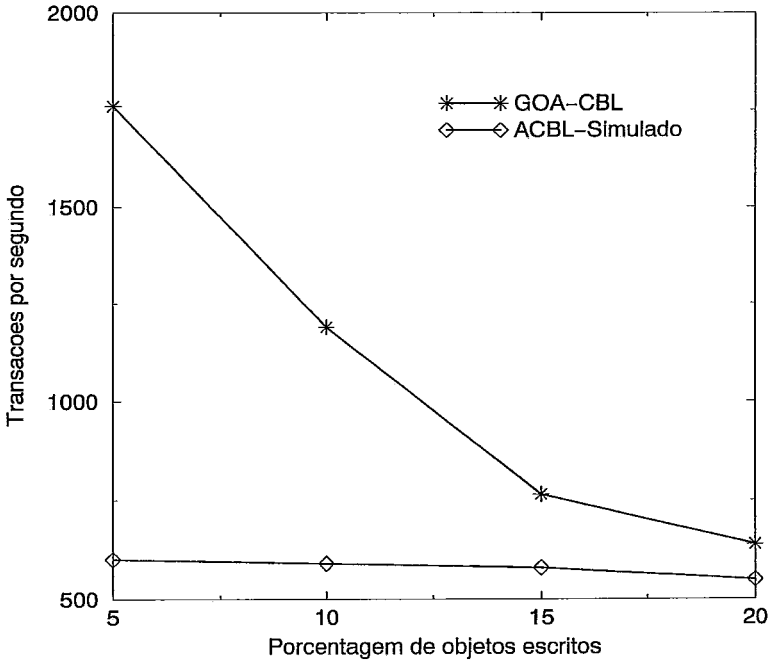

Figura 7.2: Desempenho da aplicação **privada** nos sistemas cliente-servidor GOA-CBL e ACBL-Simulado

Carga Alta Contenção

Na figura 7.4 apresentamos o desempenho da aplicação **alta contenção** nos sistemas cliente-servidor GOA-CBL e ACBL-Simulado. O gráfico compara o número de transações por segundo em relação a porcentagem de dados escritos. Esta porcentagem aumenta de 5% a 20%.

Observamos na figura que à medida em que a percentagem de escrita aumenta, que a taxa de transações por segundo de ambos os sistemas diminui.

Observamos também que a taxa de transações do sistema GOA-CBL é pouco maior que o sistema ACBL-Simulado, e que a diferença se mantém pequena a medida em que se aumenta a percentagem de escrita do sistema.

7.1.3 Discussão

Apesar de diferenças tão importantes os resultados obtidos entre o sistema GOA-CBL real e o sistema ACBL-Simulado pode-se afirmar que os desempenhos são próximos.

Como o desempenho do sistema GOA-CBL gerou resultados superiores ao desem-

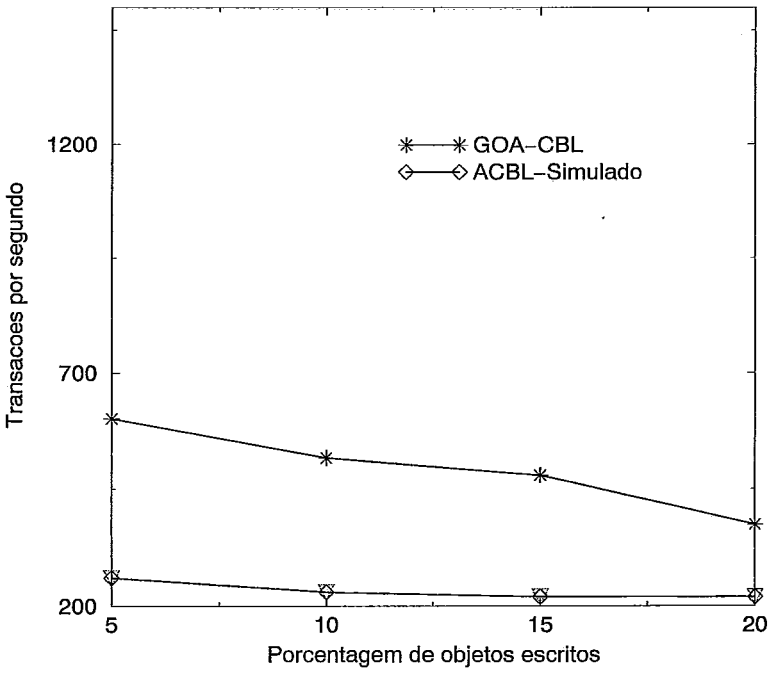

Figura 7.3: Desempenho da aplicação **compartilhada** nos sistemas cliente-servidor GOA-CBL e ACBL-Simulado

penho do sistema ACBL-simulado, podemos concluir que existem indicações de que p sistema cliente-servidor GOA-CBL desenvolvido possui um desempenho aceitável, e compatível com a proposta CBL.

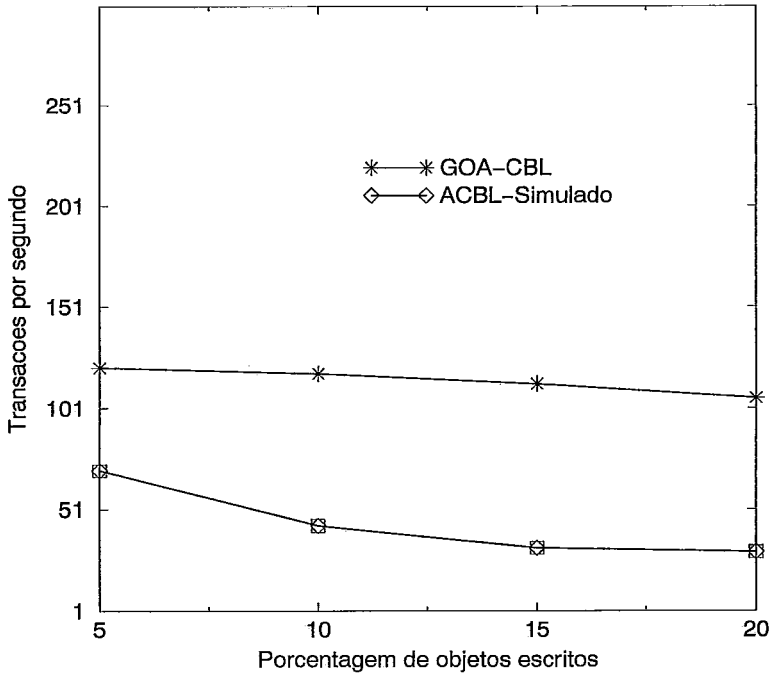

Figura 7.4: Desempenho da aplicação **alta contenção** nos sistemas cliente-servidor GOA-CBL e ACBL-Simulado

Capítulo 8

Avaliação de Desempenho do Sistema SGBD Cliente-Servidor DSMIO

8.1 Análise de Resultados

Nesse capítulo iremos analisar o desempenho dos sistemas cliente-servidor GOA-CBL (ver 6.2) e GOA-DSMIO (ver 6.3) para as cargas **Privada, Compartilhada e Alta Contenção.**

8.1.1 Carga Privada

Na figura 8.1 apresentamos o desempenho da aplicação de carga **privada** nos sistemas cliente-servidor GOA-CBL e GOA-DSMIO.

Como pode ser observado, á medida em que a percentagem de escrita aumenta, a taxa de transações por segundo de ambos os sistemas diminui, embora a taxa de transações do sistema GOA-DSMIO seja bem maior que a do sistema GOA-CBL, e que essa diferença vai aumentando significativamente.

Isto se deve ao fato de que os sistemas apresentam desempenho similar para operações de leitura enquanto que o sistema DSMIO possui um desempenho muito superior que o sistema CBL para as operações de escrita.

A figura 8.2 apresenta o número de mensagens por operações de transação em função da porcentagem de escrita dos sistemas GOA-CBL e GOA-DSMIO. Vemos que no protocolo CBL uma operação de escrita gera um número muito grande de mensagens.

A figura 8.3 apresenta o número de *diffs* gerados durante a execução da aplicação em função da porcentagem de objetos escritos. Ela apresenta tamém o número de

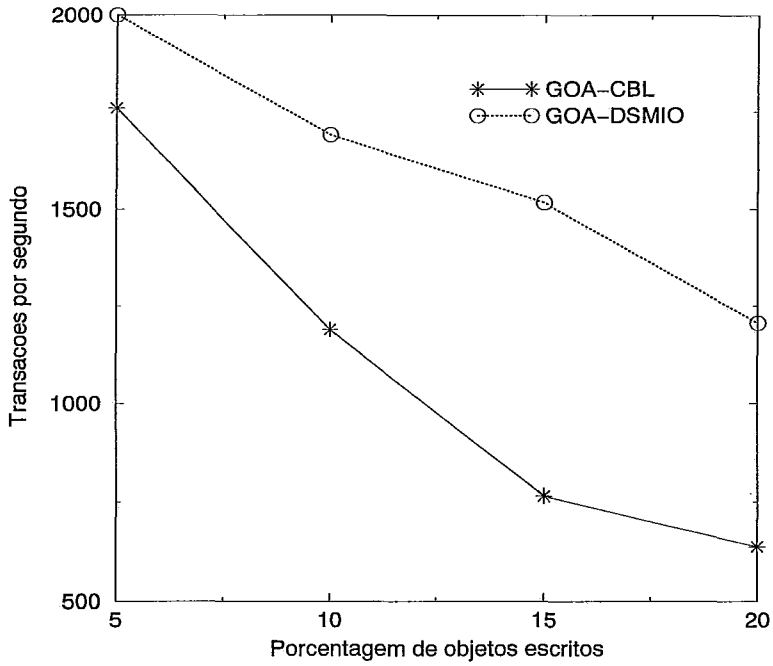

Figura 8.1: Desempenho da aplicação **privada** nos sistemas cliente-servidor GOA-CBL e GOA-DSMIO

diffs gerados para as outras duas aplicações a serem analisadas por uma questão de simplificação de gráficos. Ao compararmos a figura 8.3 com a figura 8.2, vemos que no sistema GOA-CBL o número de mensagens aumenta à medida em que a percentagem de escritas aumenta enquanto que no sistema GOA-DSMIO a porcentagem de escritas se mantém praticamente constante enquanto que a porcentagem de *diffs* aumenta. Isto mostra que no sistema DSMIO as operações de escrita são convertidas em operações de geração de *diffs*.

A tabela 8.1 apresenta os parâmetros de leitura do sistema GOA-DSMIO. As tabelas 8.2 e 8.3 apresentam, respectivamente, os parâmetros de uma operação de escrita no sistema cliente-servidor GOA-DSMIO e no sistema GOA-CBL. Vemos que os gastos produzidos para a geração de um *diff* de uma página a ser atualizada é cerca de uma ordem de grandeza menor que o gasto gerado para o envio de uma página inteira para o servidor. Vemos pelas tabelas que a geração e a recuperação de um *diff* possui um custo bem menor que o envio de uma página inteira ao servidor. Nesta aplicação os *diffs* são compostos por poucos bytes (em média 16). Sem contar que, como esta aplicação não possui compartilhamento, os *diffs* são recuperados apenas uma vez, durante a operação de *Flush de cache* do servidor.

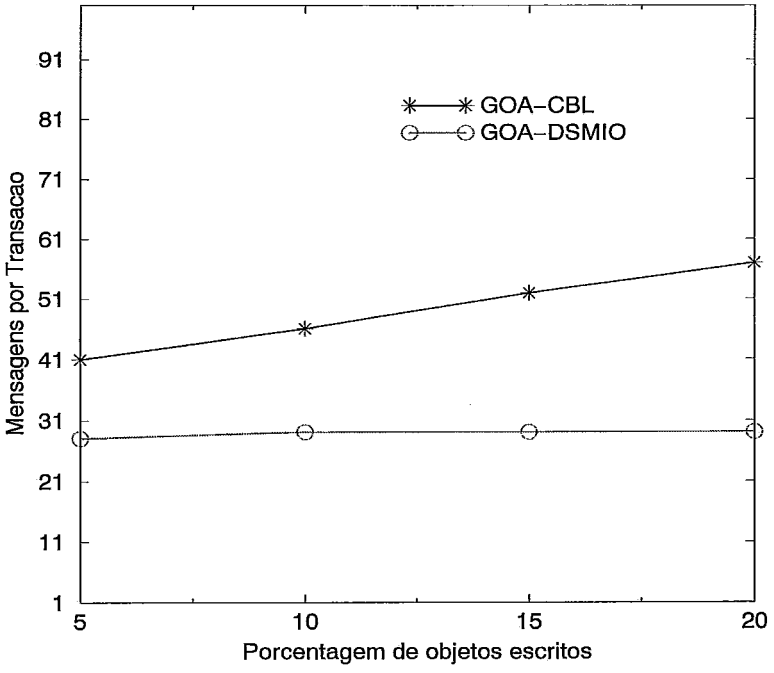

Figura 8.2: Mensagens por transação na aplicação **privada** nos sistemas cliente-servidor GOA-CBL e GOA-DSMIO

Estes resultados foram obtidos em uma rede de alto-desempenho, de 40Mbytes por segundo, onde a transmissão de mensagens executa o protocolo UDP, um protocolo que não possui controle de fluxo de mensagem e que, por isto, apresenta alto desempenho para redes como no caso do *switch*. Caso os testes venham a ser realizados em uma rede de menor desempenho, ou seja, com maior latência na transmissão das mensagens e menor banda-passante, tal como uma rede *ethernet* tipicamente adotada em rede de *workstations*, os ganhos relativos a economia no envio de mensagens serão ainda mais significativos.

Podemos observar nestas tabelas que o tempo médio de leitura e escrita em disco de ambos sistemas GOA-DSMIO e GOA-CBL são da ordem de grandeza do tempo de acesso a memória local. Isto se deve à eficiência do sistema de *prefetching* de páginas dos discos pelo sistema operacional. Neste sistema, quando ocorre um pedido de leitura a uma página armazenada em disco, o sistema traz, além da página solicitada, as próximas 100 páginas do disco, de forma que o segundo acesso ao disco acaba sendo sempre um acesso ao *buffer* local do sistema operacional. O mesmo ocorre em uma operação de escrita. Um pedido de escrita escreve a página em um *write buffer* do sistema operacional que tem o tempo de acesso da ordem

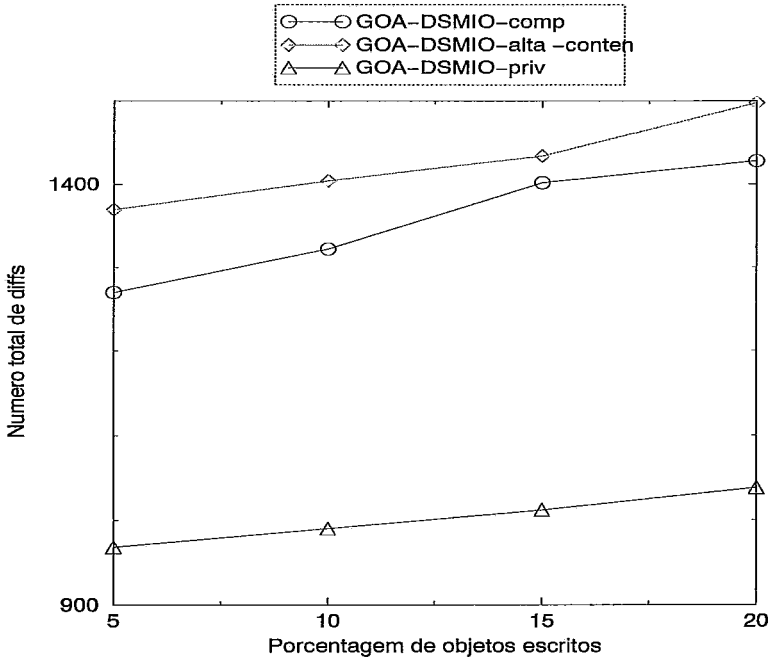

Figura 8.3: Geração de *diffs* na aplicação **privada** nos sistemas cliente-servidor GOA-DSMIO

Tabela 8.1: Parâmetros de um operação de leitura no GOA-DSMIO

Descrição	valor médio
acesso a memória local	200us
acesso ao disco local	200us
leitura remota	4000us

de um acesso a memória local e mais tarde o sistema operacional irá armazenar a página no disco.

Concluímos que as aplicações utilizadas para esta análise de desempenho não possuem padrão de E/S complexos e intensos suficientes para estressar a capacidade de E/S do sistema operacional. Para isto é necessário testar aplicações com bases maiores e padrões de acesso ao disco mais complexos.

8.1.2 Carga Compartilhada

Na figura 8.4 apresentamos o desempenho da aplicação de carga **compartilhada**. Procuramos manter o padrão da carga **Compartilhada** próximo ao da carga **Privada**. De forma que a carga **compartilhada** executa a mesma aplicação que

Tabela 8.2: Parâmetros de um operação de escrita no GOA-DSMIO

Descrição	valor médio
gera diff	400us
acesso (escrita) a memória local	200us
escrita ao disco local	200us
geração de *twin*	200us

Tabela 8.3: Parâmetros de um operação de escrita no GOA-CBL

Descrição	valor médio
envia página para servidor	4000us
acesso (escrita) a memória local	150us
escrita ao disco local	200us

a carga **privada**. A única diferença está no fato de que na carga **privada** os 20% dos dados compartilhados são apenas lidos pelos nós clientes enquanto que na carga **compartilhada** eles são lidos e escritos pelos nós clientes. Desta forma podemos observar os gastos introduzidos no sistema pelas operações de escrita a dados compartilhados.

Observamos na figura 8.4 que 'a medida em que a percentagem de escrita aumenta, a taxa de transações por segundo de ambos os sistemas diminui, porém a taxa de transações do sistema GOA-DSMIO é bem maior que o sistema GOA-CBL, e que a diferença aumenta gradativamente. Mesmo para uma pequena taxa de escrita, o sistema GOA-DSMIO apresenta quase que o dobro do número de transações por segundo que o sistema GOA-CBL. Isto se deve ao fato que a presença de carga compartilhada no sistema causa uma geração muito grande de mensagens de invalidação no sistema GOA-CBL que não acontecia na carga privada. O aumento do número de mensagens pode ser observado pela figura 8.5, que apresenta o número de mensagens geradas por operação de transação em função do número de porcentagem de objetos escritos. No sistema GOA-DSMIO, o aumento do número de *diffs*, pode ser observado pela figura 8.3, que apresenta o número de *diffs* gerados durante a execução da aplicação em função da porcentagem de objetos escritos.

Assim como na carga anterior, os gastos causados pela geração de um *diff* de uma página a ser atualizada é cerca de uma ordem de grandeza menor que o gasto para o envio da mesma página para servidor.

Nesta aplicação podemos observar também o excesso de mensagens geradas para

Tabela 8.4: Parâmetros de um operação de leitura no GOA-CBL

Descrição	valor médio
busca de uma página no servidor	200us
acesso leitura a memória local	150us
servidor le do disco local	200us
servidor busca a página remotamente	8000us

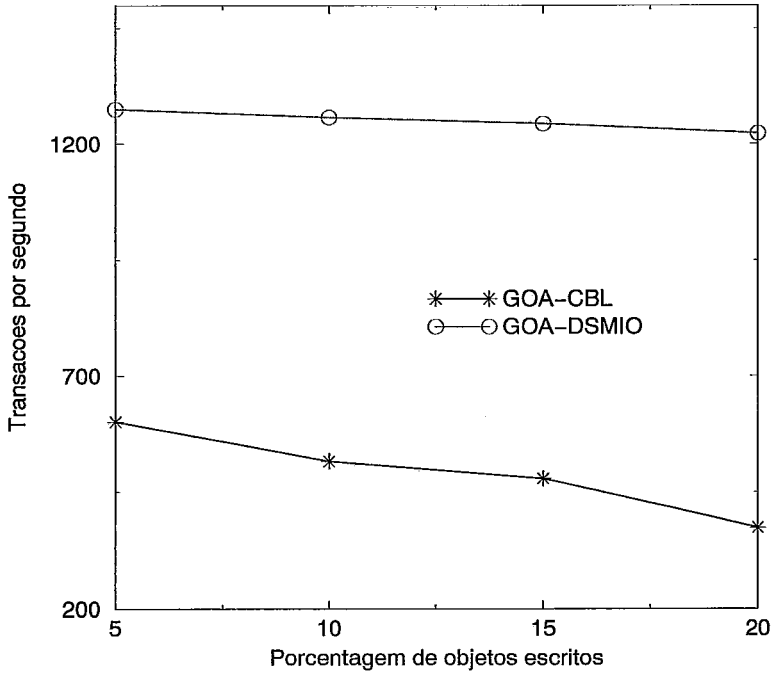

Figura 8.4: Desempenho da aplicação **compartilhada** nos sistemas cliente-servidor GOA-CBL e GOA-DSMIO

a execução da invalidação das páginas dos nós clientes que possuem cópia do dado. No sistema GOA-DSMIO, quando uma página é atualizada e o seu respectivo *diff* é armazenado, são geradas informações de escritas, i.e. *write notices* que serão transmitidas para o nó que executar a próxima operação de *lock* naquela página. Este nó irá receber esta informação em conjunto com as informações de todas as invalidações ocorridas desde a última vez em que ele executou uma operação de bloqueio (ou de barreira). Este nó irá invalidar as páginas que estão desatualizadas e o *diff* só será solicitado caso a página venha a ser acessada, de acordo com o protocolo do sistema *TreadMarks*.

A carga **compartilhada** aumenta significativamente a quantidade de geração e recuperação de *diffs*, degradando signitivamente o desempenho do sistema

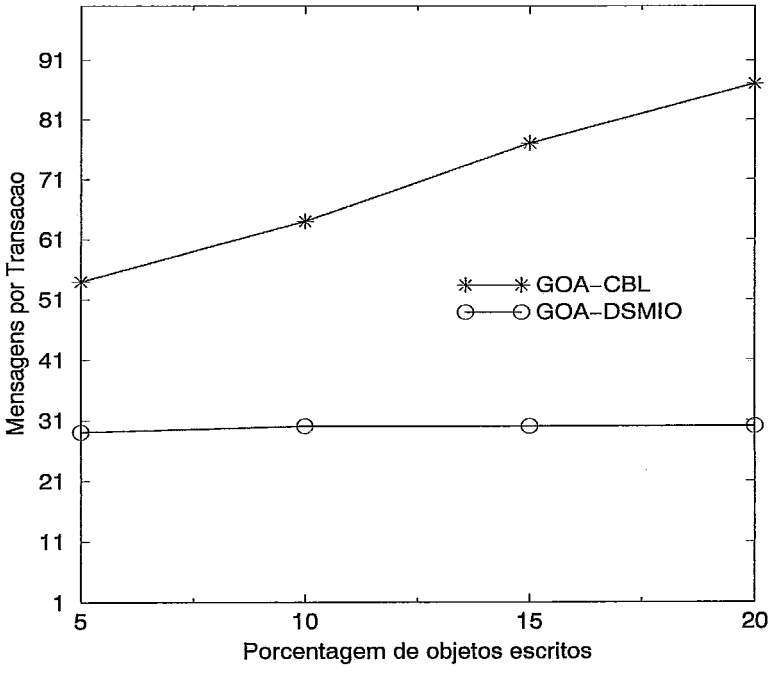

Figura 8.5: Mensagens por transação na aplicação **compartilhada** nos sistemas cliente-servidor GOA-CBL e GOA-DSMIO

GOA+DSMIO. Mesmo assim, o custo de geração e de recuperação de *diffs* é bem menor do que o envio das páginas inteiras para a atualização das páginas no servidor somadas ao custo do envio de mensagens de invalidação aos nós clientes no sistema GOA+CBL.

8.1.3 Carga de Alta Contenção

Na figura 8.6 apresentamos o desempenho da aplicação de **alta contenção** nos dois sistemas.

Procuramos manter o padrão da carga de **alta contenção** como sendo exatamente o oposto da carga **compartilhada**. Ou seja, ambos realizam a mesma aplicação, sendo que na carga compartilhada 20% dos dados escritos são compartilhados enquanto que na carga de **alta contenção** 80% dos dados escritos são compartilhados. Desta forma podemos observar os gastos introduzidos pelo excesso de mensagens nos sistemas quando se aumenta significativamente a taxa de dados compartilhados.

Observa-se que a medida em que a percentagem de escrita aumenta, a taxa de transações por segundo de ambos os sistemas diminui. Além dissso a taxa de

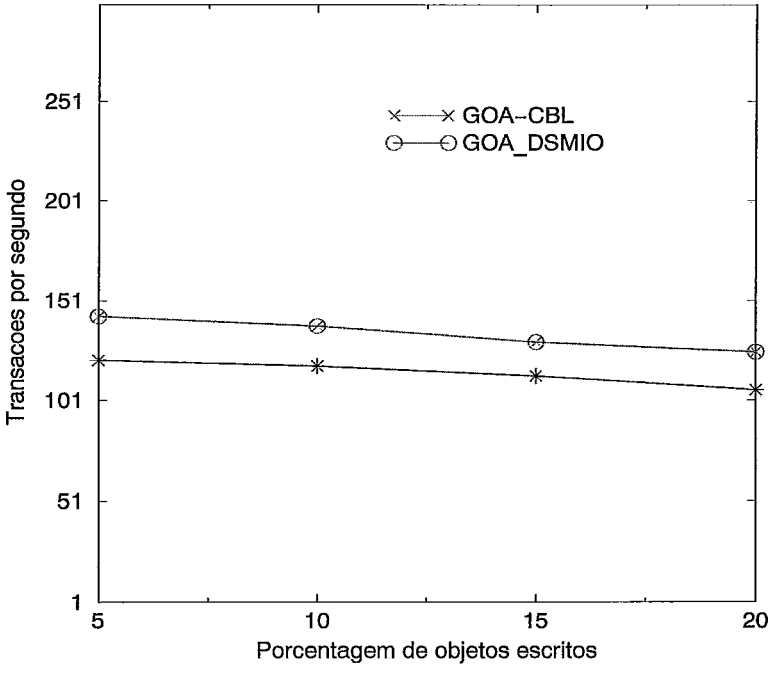

Figura 8.6: Desempenho da aplicação **alta contenção** nos sistemas cliente-servidor GOA-CBL e GOA-DSMIO

transações do sistema GOA-DSMIO é pouco maior que o sistema GOA-CBL, por volta de 25% e que a diferença aumenta muito pouco a medida em que se aumenta a percentagem de escrita do sistema.

Ambos os sistemas sofrem uma grande degradação. No caso do GOA-CBL, isto se deve ao fato que o aumento da carga compartilhada causa uma geração ainda maior de mensagens de invalidação como pode ser observado pelo figura 8.7.

No caso do GOA-DSMIO a taxa de compartilhamento aumenta consideravelmente a taxa de geração/recuperação de *diffs* pelo sistema. A figura 8.3 apresenta o número de *diffs* gerados durante a execução da aplicação em função da porcentagem de objetos escritos.

Há também um aumento considerável no tamanho das mensagens transmitidas pelos sistemas. Isto é devido ao fato de que nesta aplicação um grande número de objetos de uma mesma página são atualizados. Isto acarreta o aumento do tamanho da mensagem de *diff* sendo transmitida, em média de 300 bytes. Esse aumento inclui o número de *bytes* que foram alterados em uma página durante uma operação de transação. No pior caso, todos os *bytes* da página são alterados e o tamanho da mensagem *diff* equivale ao tamanho da página inteira, o que não é o caso desta

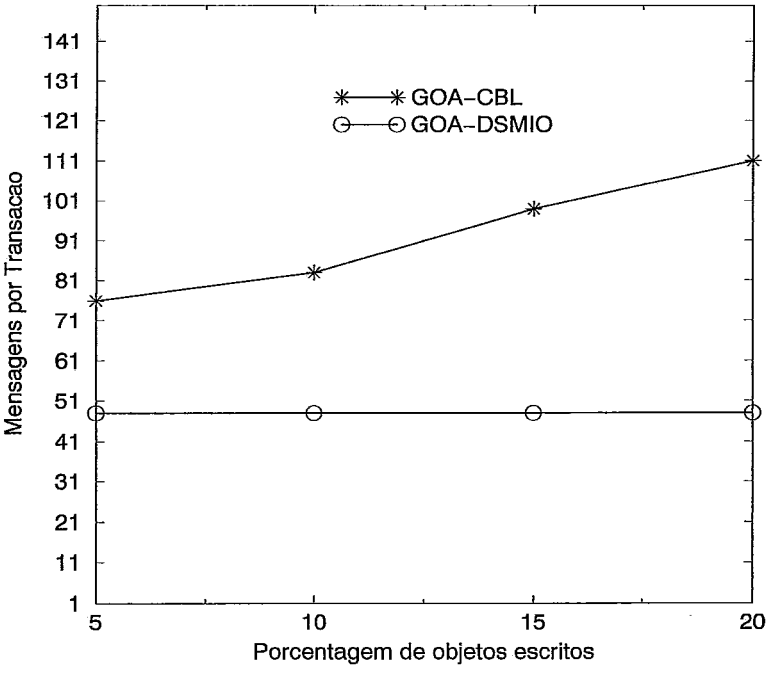

Figura 8.7: Mensagens por transação na aplicação **alta contenção** nos sistemas cliente-servidor GOA-CBL e GOA-DSMIO

aplicação.

Apesar do tamanho dos *diffs* ter aumentado significativamente, o sistema GOA-DSMIO continua a ser mais eficiente que o sistema GOA-CBL, como pode ser observado na figura 8.6.

8.1.4 Conclusões

Os resultados comprovam a eficiência mostrada no contexto de sistema de DSMIO em adotar um modelo de consistência LRC ao invés de modelos menos relaxados para a manutenção da coerência dos dados compartilhados e de empregar mecanismos de *diffs* para o armazenamento dos dados escritos no disco.

Nossos resultados mostram que os gastos gerados pela criação e posterior adaptação dos *diffs* do sistema DSMIO são suplantados pelos ganhos que esse sistema obtém através da eliminação de mensagens de manutenção da coerência dos dados.

Capítulo 9

Avaliação do Sistema paralelo cliente-servidor baseado em DSMIO

Em termos de modelo de programação, um sistema paralelo de banco de dados atua como um sistema servidor de banco de dados para múltiplas aplicações, de maneira similar a uma organização de redes de estações de trabalho do tipo cliente-servidor [6]. A diferença fundamental entre estes dois sistemas está no fato de que um sistema paralelo de banco de dados atende não apenas às funções de banco de dados e as interfaces cliente-servidor como também a funções gerais tais como funções relacionadas com a gerência do banco de dados.

Com relação ao modelo de memória mais adequado para a exploração de paralelismo em SGBDOO, alguns autores como Valduriez [19] consideram ser mais difícil a utilização da arquitetura de memória distribuída devido à complexidade de conciliar ao mesmo tempo a distribuição e o agrupamento dos dados. Segundo o autor, tanto a arquitetura de memória compartilhada como a arquitetura de disco compartilhado permitem adotar soluções tradicionais de agrupamento utilizadas em sistemas seqüenciais. Outros autores, como DEWITT [20] apresentam as vantagens da arquitetura de memória distribuída (expansibilidade, custos mais reduzidos) adotando-o no SGBDOO SHORE, sem no entando apresentarem resultados em situações desfavoráveis ao paralelismo.

Por outro lado, as consultas em SGBDOOs tendem a envolver uma navegação entre classes. Desta forma, uma estratégia efetiva para distribuição de objetos pelos diversos nós em um ambiente de memória distribuída é crucial para que se obtenha um bom desempenho. Uma vez iniciada em um nó, a consulta deve percorrer os

relacionamentos preferencialmente no âmbito do próprio nó. Do contrário, o acesso a objetos residentes em outros nós implicará em comunicação entre os diversos processadores e podem comprometer o ganho com o paralelismo. Portanto, a estratégia de distribuição deve procurar reduzir a possibilidade de comunicação entre os nós, mas ao mesmo tempo permitir que sejam exploradas diferentes formas de paralelismo no processamento de uma consulta.

Podemos ver que existe um conflito entre estes dois objetivos. Se por um lado a redução de comunicação entre os diversos processadores favorece que se tenha a base de dados distribuída por um pequeno número de nós, para se alcançar um alto grau de paralelismo é necessária uma distribuição da base de dados por um grande número de processadores.

Várias arquiteturas foram propostas e desenvolvidas com o objetivo de unir escalabilidade com desempenho, conforme apresentamos na seção 4.2.1, dentre elas podemos destacar a arquitetura NUMA.

Sistemas multiprocessadores de memória compartilhada com arquitetura NUMA são propostas como a plataforma de escolha para a execução de sistemas comerciais de banco de dados [81]. Comparados a sistemas de banco de dados convencionais, esses sistemas oferecem uma melhor relação custo/desempenho em um sistema escalável com um ambiente de computação de programação mais simples.

Arquiteturas NUMA baseadas em *hardware* DSM são utilizadas para combinar escalabilidade com simplicidade de programação. O objetivo está em fornecer um modelo de programação de memória compartilhada e todos os seus benefícios em uma arquitetura paralela escalável.

Por outro lado, sistemas paralelos construídos com computadores e redes de comunicação comerciais, chamados *off-the-shelf* , tais como *clusters beowulf* [95] , oferecem uma plataforma escalável de baixo custo. O sistema DSMIO possibilita o desenvolvimento de sistemas paralelos de banco de dados cliente-servidor exatamente nesta classe.

Neste capítulo, iremos avaliar o desempenho do sistema DSMIO no desenvolvimento de um sistema SGBDOO paralelo cliente-servidor.

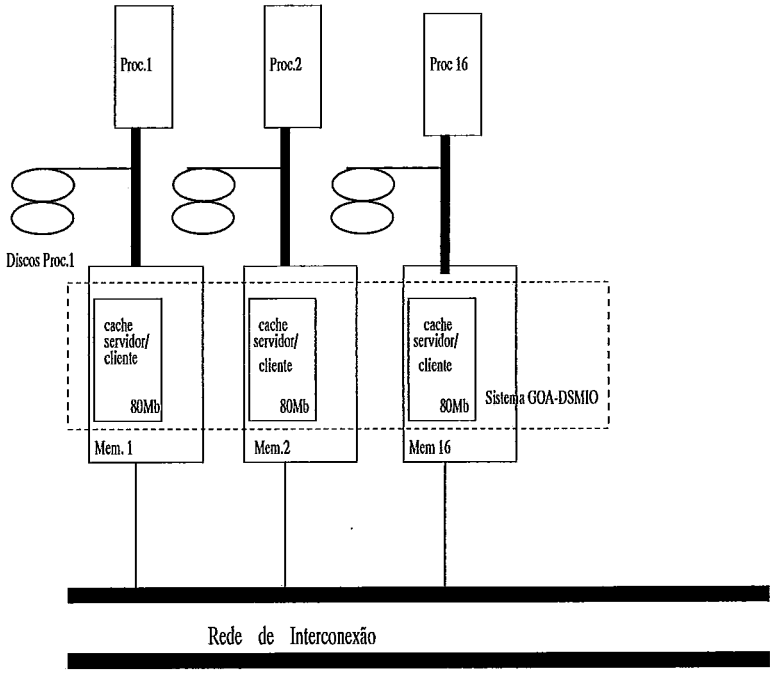

Figura 9.1: O sistema paralelo cliente-servidor GOA-DSMIO.

9.1 Metodologia

Assim como nos testes anteriores, avaliamos o nosso sistema com o GOA [15, 2], que possui uma cache de 4Mbytes.

Desenvolvemos dois sistemas SGBDOO cliente-servidor paralelos, baseados em GOA-CBL (figura 6.2) e GOA-DSMIO, (figura 6.3). A diferença está no fato de que nestes novos sistemas a base de dados é fragmentada entre os discos de todos os processadores do sistema de forma que todos os processadores exercem ao mesmo tempo a funções de servidor de dados e de cliente.

Os dois sistemas estão ilustrados nas figuras 9.1 e 9.2

- A figura 9.1 apresenta o sistema SGBD00 paralelo cliente-servidor GOA+DSMIO com 16 processadores. Na combinação de GOA +DSMIO, as caches locais de cada processador são substituídas pela cache compartilhada gerenciada pelo sistema DSMIO. A base de dados é fragmentada de forma *round robin* circular pelos 16 processadores do sistema. De forma a que cada processador possa ser *home* de um setor da base de dados.

- A figura 9.2 apresenta o sistema SGBDOO paralelo cliente-servidor GOA-CBL com 16 processadores. Nesse sistema, cada processador possui uma cache local

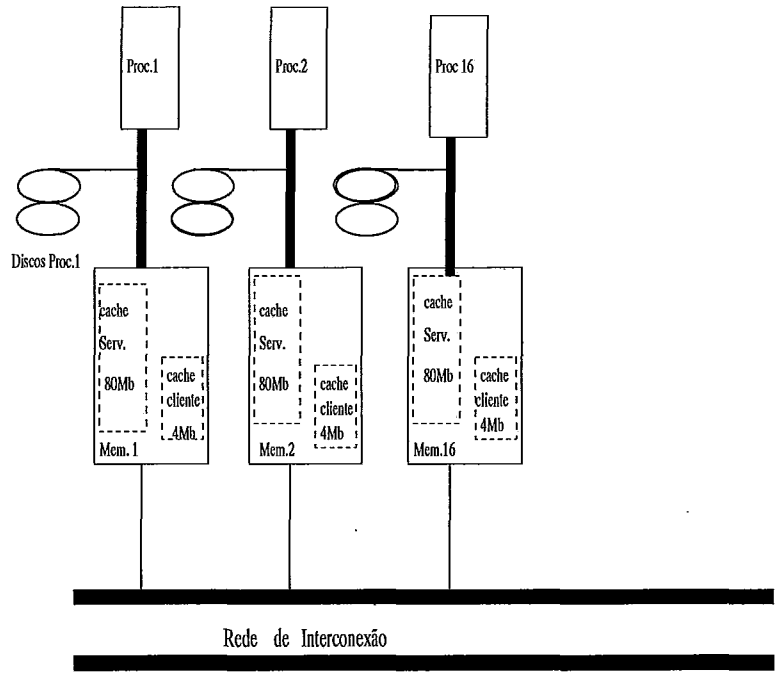

Figura 9.2: O sistema paralelo cliente-servidor GOA-CBL.

gerenciada pelo sistema SGBD GOA e uma cache servidora gerenciada pelo processo servidor de dados do disco que mantêm a coerência dos dados segundo o protocolo do algoritmo CBL. Um processo cliente armazena os seus dados na sua cache local e solicita os dados para o processo servidor de disco. A base de dados é fragmentada pelos 16 processadores do sistemade forma *round robin* circular pelos 16 processadores do sistema. De forma a que cada processador possa ser *home* de um setor da base de dados.

Para avaliarmos o desempenho dos sistemas SGBDOO paralelos cliente-servidor, doravante denominados simplesmentes GOA-DSMIO e GOA-CBL, nós realizamos testes similares aos desenvolvidos no capítulo 5, onde implementamos a aplicação de consulta de suporte à decisão **T3**, pertencente ao *Benchmark 007* [4] paralelizada segundo o modo de programação SPMD, com operações de sincronização para evitar acessos conflitantes aos objetos compartilhados, e utilizamos a base de tamanho médio (100 Mbytes). Nesta aplicação todos os objetos *assembly parts* percorridos pela aplicação são atualizados conforme a descrição apresentada no capítulo 4 e os servidores do sistema repartem a base de dados de forma " round-robin".

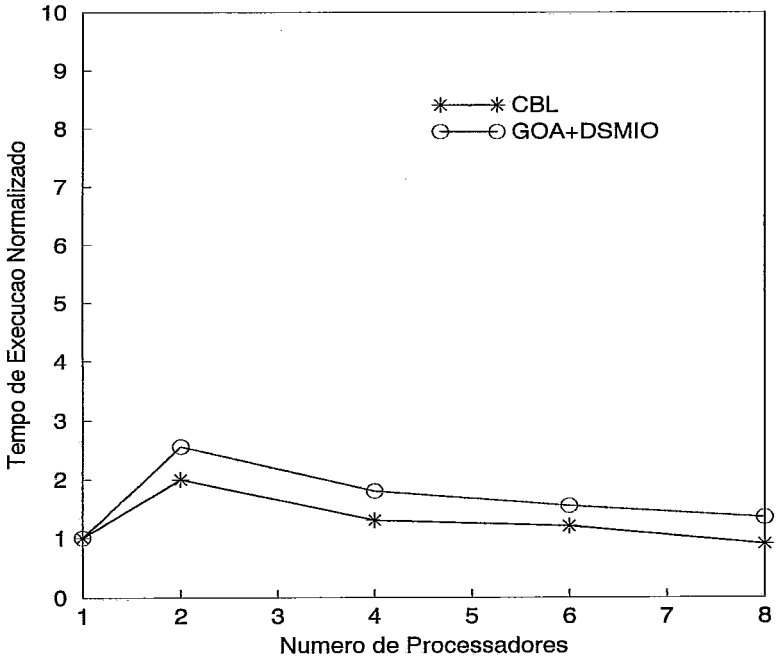

Figura 9.3: Desempenho das leituras (50% remotas) em GOA+DSMIO e GOA+CBL

9.2 Distribuição de dados desbalanceada

Realizamos um primeiro teste com uma distribuição da base de dados desbalanceada, onde cada processador acessa 50% dos seus dados remotamente e 50% dos seus dados localmente.

A figura 9.3 apresenta as curvas de tempo de execução de T3 executando apenas operações de leitura nos dois sistemas. A figura confirma que sistemas distribuídos apresentam desempenhos ruins quando existe desbalanceamento de carga. Podemos observar que ambos os sistemas possuem um desempenho muito ruim com 2 processadores. Isto se deve ao fato de que desbalanceamento de carga foi maior para 2 processadores. Neste experimento a base foi fragmentada de tal forma que 90% dos dados acessados estava localizados no processador 2. Já para os outros experimentos, de 4 a 8 processadores, o desbalanceamento é de 50%, onde 50% dos dados acessados remotamente são distribuídos de forma circular *round-robin* pelos processadores do sistema.

Podemos observar que para aplicações que possuem apenas operações de leitura, o sistema GOA+DSMIO possui um desempenho um pouco inferior ao do sistema GOA+CBL.

Isto é devido ao custo introduzido por DSMIO para transferir páginas para sua

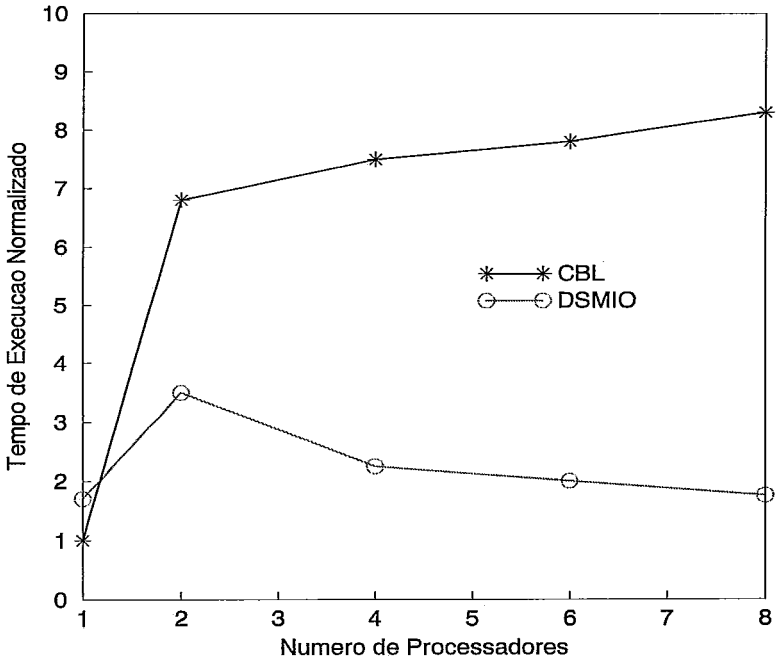

Figura 9.4: Desempenho das escritas (100% remotas e sem compartilhamento) em GOA+DSMIO e GOA+CBL

cache compartilhada. Este custo corresponde ao tempo de execução da função para desabilitar as interrupções de proteção de página do sistema operacional, para que o sistema DSMIO possa gerar diffs apenas quando ocorrer operações de escrita na cache compartilhada. Num sistema DSMIO adaptativo para a leitura, esse custo deverá ser minimizado como mostrado adiante.

9.2.1 Análise dos Resultados

Nesta seção, comparamos o desempenho dos dois sistemas executando operações de escrita em objetos compartilhados, onde 100% dos objetos escritos são compartilhados. Nos concentramos apenas nos aspectos de consistência de memória dos sistemas e nos ganhos gerados pelos mecanismos de *diffs*.

Os ganhos gerados pelo modelo LRC são devido à redução na quantidade de mensagens e aos ganhos gerados pelo mecanismo de *diffs*, o qual reduz o tamanho das mensagens e o espaço de armazenamendo de dados da memória local.

Na figura 9.4, apresentamos as curvas de tempo de execução de T3 em GOA+CBL e GOA+DSMIO em função do número de processadores. Os tempos estão normalizados com relação ao tempo de execução da aplicação T3 no GOA+CBL com 1 processador. Verificamos que o tempo de execução do siste-

ma GOA+DSMIO é bem mais eficiente que o sistema GOA+CBL, mas que, devido ao desbalanceamento de carga, ambos os sistemas são mais eficientes com apenas 1 processador.

Apesar de não apresentar *speed up*, observamos na figura 9.4 que GOA+DSMIO apresenta ganhos crescentes de desempenho em relação ao GOA+CBL a medida em que aumentamos o número de processadores do sistema. Isso se deve à grande redução obtida por GOA+DSMIO no tempo de recuperação de um dado compartilhado e no tempo de propagação das modificações realizadas no dado compartilhado.

A redução obtida no tempo de recuperação do dado compartilhado é devido à GOA+DSMIO precisar somente receber e aplicar os *diffs* correspondentes à este dado, enquando que GOA+CBL necessita da última versão do dado executada pela operação de escrita no disco.

A redução obtida no tempo de propagação das modificações realizadas no dado compartilhado é conseqüência da diferença dos modelos de consistência de memória adotados pelos dois sistemas. Mais especificamente, GOA+CBL envia uma mensagem de atualização do dado para o servidor do mesmo e mensagens de invalidação para os outros processadores do sistema que compartilham o mesmo dado. O processador que escreveu o dado deve esperar *acknowledgements* de todos os processadores que receberam invalidações e um *acknowledgement* do servidor.

GOA+DSMIO diminui drasticamente o número de mensagens e a quantidade de dados trocados, atrasando a propagação das informações de coerência até o momento do próximo pedido de *lock*. Nesse momento, somente o processador que requisitou o *lock* recebe informações de coerência sobre os dados compartilhados. Dessa forma, GOA+DSMIO evita o excesso de tráfego gerado por mensagens de invalidação e ao mesmo tempo de espera pelo *acknowledgements* assim como o tempo de espera pela escrita ao disco do servidor. Na verdade, a saída de uma seção crítica em GOA+DSMIO envolve apenas a criação de *diffs* e seu armazenamento em memória principal, não gerando, portanto, nenhum tipo de comunicação pela rede.

Além disso, GOA+DSMIO também reduz o tamanho das mensagens transmitidas já que atualiza os dados através dos mecanismos de *diffs*.

Concluímos que o sistema SGBDOO paralelo cliente-servidor GOA+DSMIO é um sistema menos sensível a um ambiente com carga desbalanceada que o sistema SGBDOO paralelo cliente-servidor GOA+CBL para aplicações que necessitam atualizar dados compartilhados.

9.3 DSMIO Adaptativo para Acessos de Leitura

Para melhorar o desempenho das leituras de DSMIO, modificamos essas operações de forma que toda página lida seja armazenada inicialmente em uma *cache* local. Além disso, o servidor armazena também a informação sobre o grau de compartilhamento da página. Caso a página tenha se tornado compartilhada por mais de um processador cliente, o servidor irá transferir esta página da *cache* local para a *cache* compartilhada para em seguida enviar a página para o processador cliente em conjunto com a informação sobre qual página compartilhada ela deverá ser armazenada, conforme o protocolo de leitura DSMIO. Por final, o servidor enviará um pedido de transferência de *cache* local para *cache* compartilhada para o nó cliente que possui cópia desta página local. Podemos ver que este novo protocolo (figura 9.5), DSMIO/AL (adaptativo para a leitura), melhora o desempenho de leituras à páginas não compartilhadas, mas introduz gastos extras na leitura de páginas compartilhadas.

Na próxima seção, realizamos um série de experimentos com DSMIO/AL para avaliar o seu desempenho sob várias cargas de trabalho. Todas as experiências apresentam uma fragmentação da base e uma repartição de objetos *Assembly Parts* entre os processadores do sistema de tal forma que cada processador, ao executar a aplicação **T3**, encontra 90% das páginas na sua base local e requisita 10% remotamente.

9.4 Desempenho de aplicações que só executam operações de leitura

Voltando à figura 9.5, quando todas as páginas lidas remotamente são privadas, ou seja, quando não existe compartilhamento de dados, DSMIO/AL possui um melhor desempenho que DSMIO, mas que no entanto continua tendo um desempenho inferior ao de CBL. A figura 9.6 apresenta o número de mensagens da aplicação T3 em função do número de processadores. Podemos observar que o número de mensagens transmitidas em GOA+DSMIO é muito inferior ao número de mensagens transmitidas em GOA+CBL. GOA+CBL transmite muito mais pelo fato de sua *cache* local ter apenas 4Mbytes. Desta forma, uma mesma página pode ser solicitada repetidamente ao servidor, enquanto que no sistema GOA+DSMIO devido à presença de uma única *cache* compartilhada, qualquer página remota é solicitada

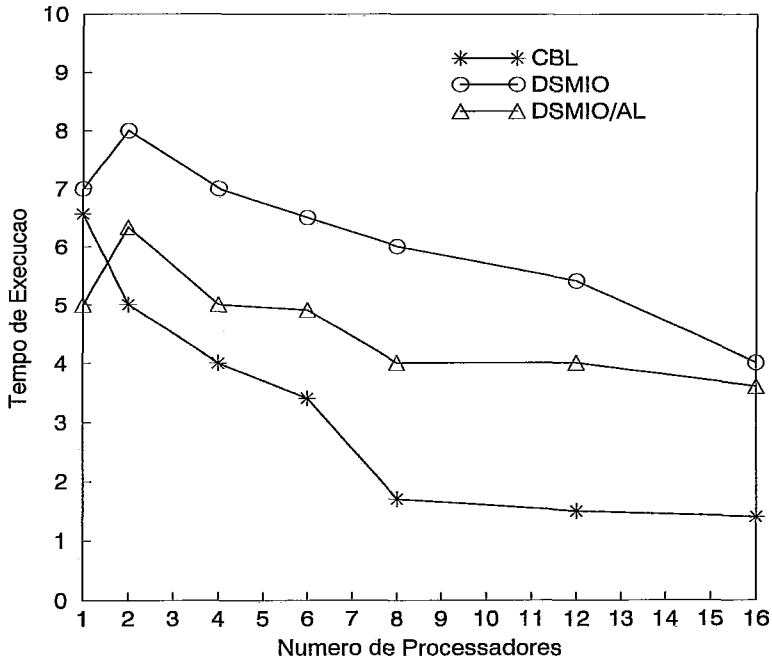

Figura 9.5: Desempenho das leituras (100% remotas/sem compartilhamento) em GOA : CBL x DSMIO x DSMIO/AL

apenas uma vez ao servidor. Apesar de trocar menos mensagens, o desempenho de DSMIO/AL é inferior ao de CBL. O problema está no custo das mensagens remotas de DSMIO/AL. Mais especificamente, verificamos que os custos provocados pela contenção do *lock* de acesso à *cache* compartilhada.

No sistema GOA+DSMIO, a *cache* compartilhada pode ser acessada e atualizada ao mesmo tempo pelo processo que atende os pedidos remotos dos nós clientes e pelos processos locais do processador. Desta forma, é necessário proteger com um *lock* o acesso à *cache* compartilhada por mais de um processo simultaneamente.

Para evitarmos estes custos extras modificamos o protocolo da seguinte forma:

- Um nó servidor, ao receber um pedido remoto, irá buscar a página solicitada na área da sua *cache* local, caso o dado não esteja disponível, ele será lido do disco. A seguir o dado será enviado para o cliente em conjunto com a informação de que o dado está sendo lido pela primeira vez, e esperar a resposta do nó contendo a informação sobre em qual página da memória compartilhada ele foi armazenado.

- Um nó cliente, ao receber uma página remota, irá verificar se a página já foi enviada anteriormente para outro cliente, caso isto não tenha ocorrido, ele

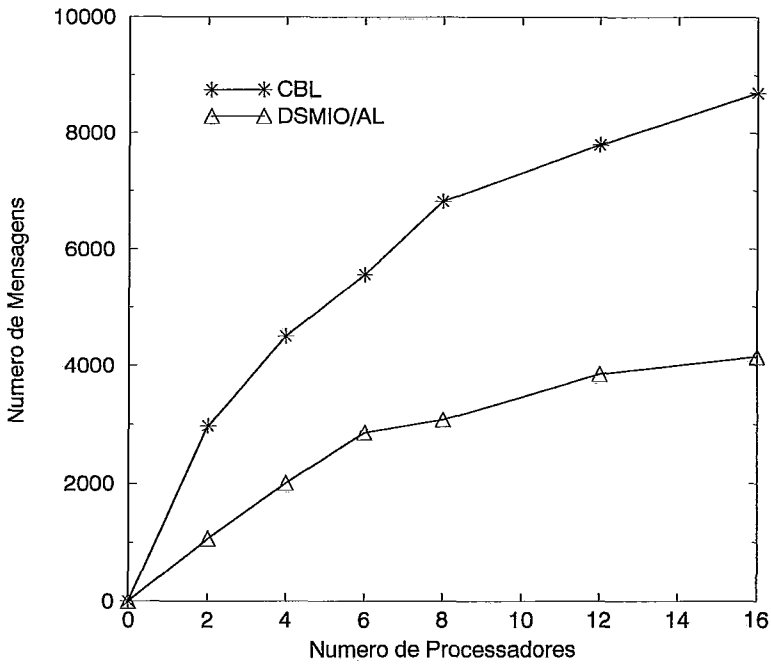

Figura 9.6: Número de mensagens em GOA : CBL x GOA-DSMIO/AL

irá armazenar a página na sua área de memória compartilhada. e enviar de volta para o servidor uma informação contendo o número desta página, caso contrário, ele irá armazenar na página informada pelo servidor.

- Um processo local, ao necessitar de ler uma página que não está na sua área compartilhada, irá solicitar o dado ao nó servidor da mesma forma que o processo remoto.

Desta forma, apenas o processo servidor acessará a *cache* local e apenas o processo cliente acessará a *cache* compartilhada. E as variáveis de *lock* de acesso às *caches* podem ser eliminadas.

Teste com leitura a objetos privados

Refizemos o teste apresentado na figura 9.5 e obtivemos o gráfico da figura 9.7. Comparando as duas figuras podemos verificar que, de fato, sem a contenção da variável de *lock* o desempenho do sistema DSMIO/NOVO melhora muito para dados remotos, conforme esperado, mas vemos que o desempenho para um processador voltou a ser igual ao desempenho do GOA+DSMIO, pois agora toda página é armazenada na memória local, conforme era executado inicialmente no sistema GOA+DSMIO. A *cache* local passou a ser empregada apenas para o acesso do processo servidor

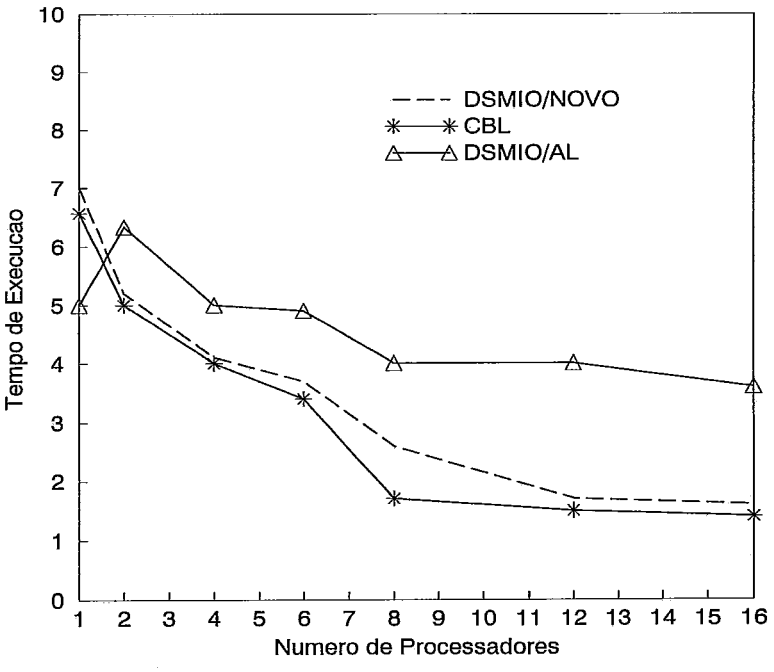

Figura 9.7: Desempenho de leitura (sem compartilhamento) em GOA : CBL x DSMIO/AL x DSMIO/NOVO

de dados, ou seja, para enviar páginas aos processos clientes, assim como é feito no sistema GOA+CBL.

Teste com leitura a páginas com objetos compartilhados

Testamos o seu desempenho para uma aplicação onde introduzimos objetos compartilhados, ou seja, para uma aplicação T3 onde 70% das páginas lidas são privadas e 30% são compartilhadas.

Na figura 9.8 apresentamos o tempo de execução das aplicação em função do número de processadores introduzindo o desempenho da nova versão GOA+DSMIO/NOVO. Podemos observar que o tempo de execução de GOA+CBL é aproximadamente igual ao tempo de execução dessa nova versão.

Ao introduzirmos páginas compartilhadas no sistema, o desempenho do sistema GOA+CBL não se altera já que as páginas compartilhadas são armazenados na *cache* da mesma forma que páginas privadas, os gastos extras são gerados pelas escritas às páginas compartilhadas.

Já na nova versão GOA+DSMIO/NOVO, mesmo que não exista escrita a páginas compartilhadas, elas são transferidas para a *cache* compartilhada, gerando novos custos ao sistema mas que são desprezíveis diante dos gastos gerados pela contenção

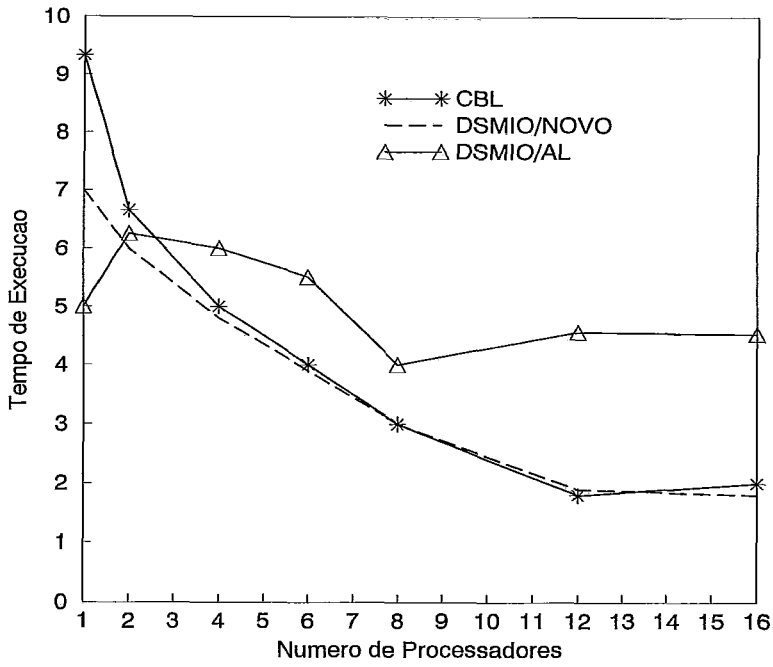

Figura 9.8: Desempenho de leitura (com compartilhamento) em GOA : CBL x GOA-DSMIO/AL x DSMIO/NOVO.

da variável de *LOCK* do sistema GOA+DSMIO/AL.

Concluímos que a separação das *caches* locais e compartilhadas de forma que elas passem a ser acessadas por apenas um processo por vez sem a necessidade de introduzirmos variáveis de *LOCK* torna o desempenho do sistema GOA+DSMIO para a leitura muito próximo ao do sistema GOA+CBL.

9.5 Ambiente com carga balanceada

A seguir apresentamos experimentos com aplicações que executam atualizações a base de dados onde 90% das páginas são acessadas localmente e 10% remotamente. E a fragmentação da base é tal que os 10% dos dados remotos são distribuidos pelas bases de forma circular *round robin.* É importante observar que a análise a ser apresentada a seguir é com o GOA+DSMIO/AL, a versão sem as otimizações de *LOCK*, cujo desempenho de leitura é mostrado na figura 9.9. A versão DSMIO/NOVO está em desenvolvimento para operações de escrita. Neste experimento, a *traversal* T3 atualiza 100% dos objetos *Atomic Part.*

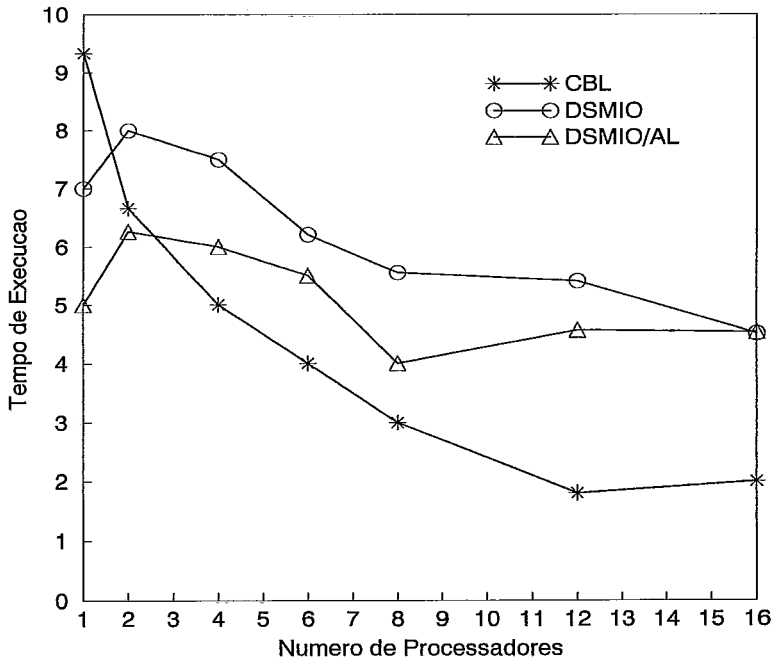

Figura 9.9: Desempenho de leitura (com compartilhamento) em GOA : CBL x GOA+DSMIO x GOA-DSMIO/AL.

9.5.1 Aplicação com escritas aos objetos compartilhadas

Neste experimento nós introduzimos operações de escrita aos objetos *atomic parts*, onde 70% dos objetos são privados e 30% dos objetos são compartilhados.

A figura 9.11, apresenta o desempenho dos sistemas GOA+CBL e GOA+DSMIO/AL para a aplicação traversal T3 onde 30% dos objetos compartilhados são atualizados.

Como pode ser visto, o desempenho do sistema GOA+DSMIO/AL se mantém praticamente constante com o aumento do número de processadores, enquanto que no sistema GOA+CBL o desempenho diminui a medida em que aumenta o número de processadores, devido não só ao excesso de mensagens enviadas pelo protocolo CBL, mas principalmente devido à contenção dos *locks* de escrita.

A figura 9.12 apresenta o número de mensagens e de *diffs* transmitido por cada sistema. Ela revela que o protocolo de coerência de DSMIO/AL é potencialmente escalável, ou seja, o número de mensagens aumenta muito pouco com o aumento do número de processadores no sistema.

A figura 9.13 apresenta o desempenho dos sistemas GOA+CBL GOA+DSMIO/AL onde 70% dos objetos privados são atualizados.

Assim como no experimento anterior, podemos observar que o desempenho do

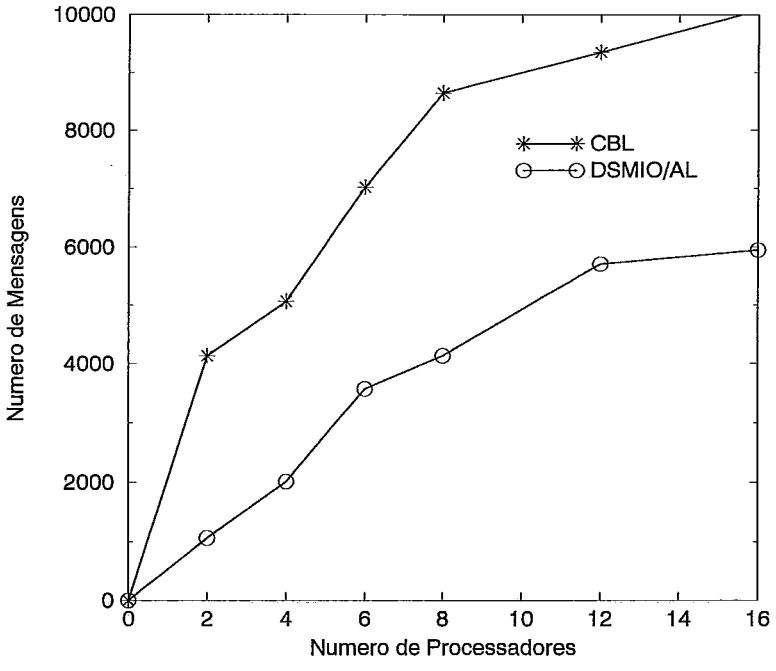

Figura 9.10: Número de mensagens de leitura (com compartilhamento) em GOA : CBL x GOA-DSMIO/AL.

sistema GOA+DSMIO/AL se mantém praticamente constante com o aumento do número de processadores, enquanto que no sistema GOA+CBL o desempenho diminui à medida em que se aumenta o número de processadores. É importante observar que apesar de serem executadas escritas apenas a dados privativos, que mesmo assim ocorre um aumento significativo do n'umero de mensagens no sistema,degradando o seu desempenho, ver 9.14. Isto é devido ao falso compartilhamento entre os dados causado pelo fato de os objetos estarem armazenados em páginas de 2kbytes.

Da mesma figura 9.14, que apresenta o número de mensagens transmitidas por cada sistemas, nós podemos observar que com o aumento do número de processadores, o sistema DSMIO gera poucas mensagens para manter a coerência dos dados, o que mostra uma escalabilidade potencial do protocolo de coerência de *cache* de DSMIO/AL.

9.5.2 Conclusões

O sistema GOA+DSMIO/AL apresenta melhor desempenho do que GOA+CBL para aplicações onde ocorre compartilhamento de objetos seguidos de atualizações, sejam estes objetos compartilhados ou não. A principal implicação é a eficiência do protocolo de coerência de *cache* de GOA+DSMIO que reduz drasticamente a

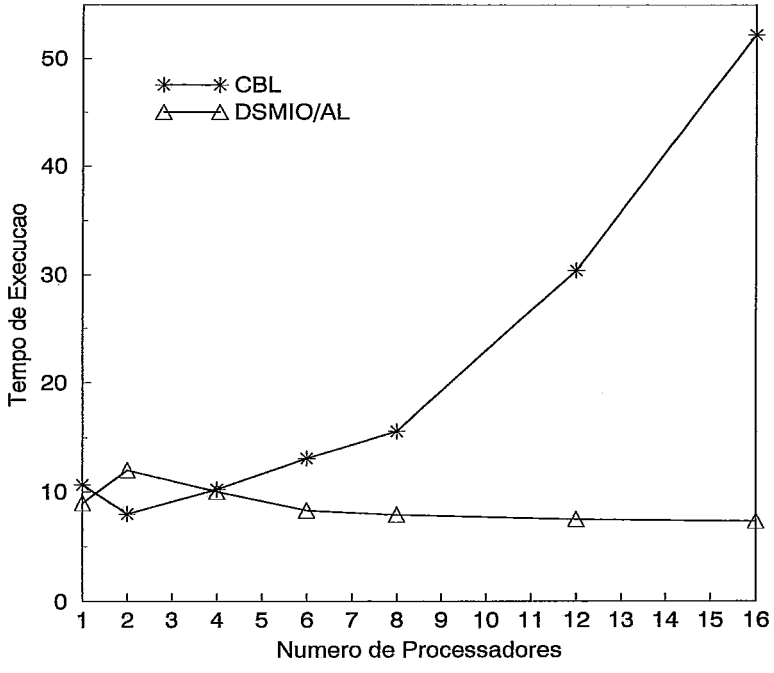

Figura 9.11: Desempenho de escrita compartilhada (30%) de GOA : CBL x GOA-DSMIO/AL.

comunicação.

Entretanto é necessário melhorar o desempenho do sistema DSMIO para leitura de páginas remotas, de forma a que este sistema venha a apresentar um melhor desempenho para aplicações que executem apenas operações de leitura a dados compartilhados.

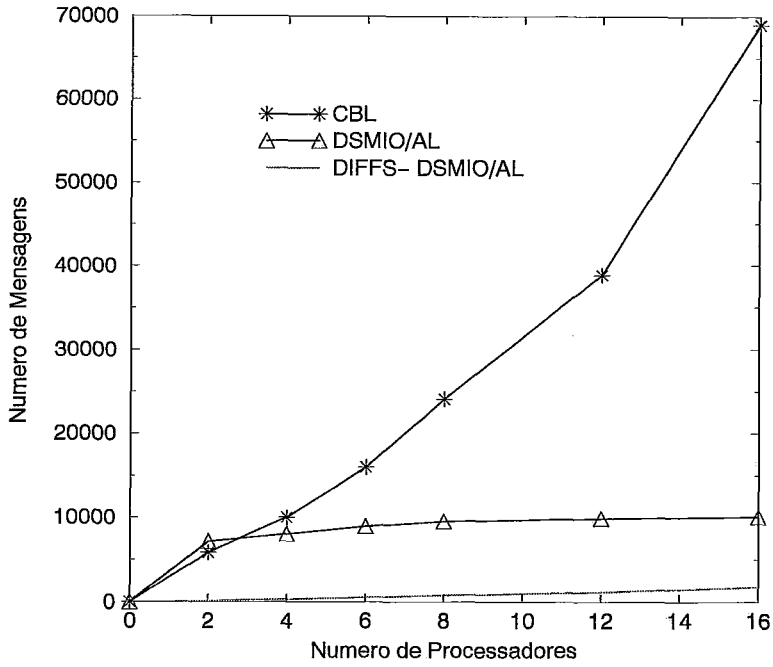

Figura 9.12: Número de mensagens em escrita compartilhada (30%) em GOA : CBL x DSMIO/AL e de *diffs* em DSMIO/AL.

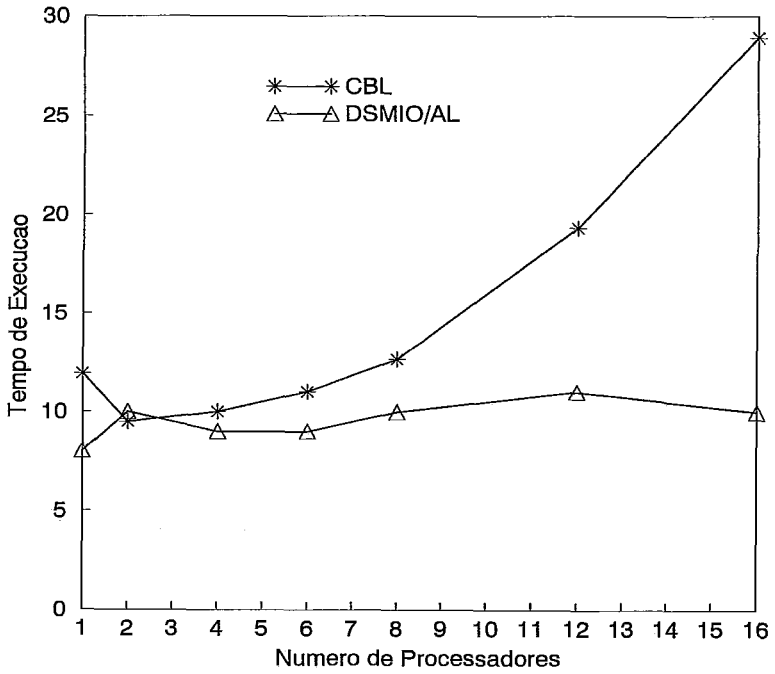

Figura 9.13: Desempenho de escrita privada (70%) com leitura compartilhada(30%) em GOA : CBL x GOA+DSMIO/AL.

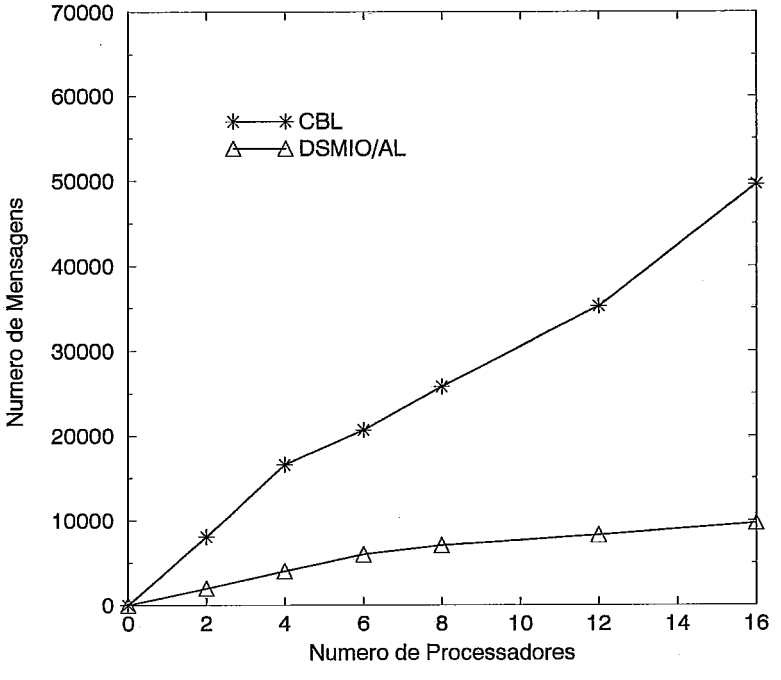

Figura 9.14: Número de mensagens de escrita privada (70%) com leitura compartilhada(30%) GOA : CBL x DSMIO/AL.

Capítulo 10

Trabalhos Relacionados

10.1 Trabalhos Relacionados

O sistema DSMIO envolve conceitos de memória compartilhada distribuída (sistemas de memória global, cacheamento cooperativo, coerência de caches e consistência de dados) na otimização de operações de E/S. Assim, nesta seção discutimos os trabalhos relacionados a cada um desses aspectos do sistema DSMIO.

10.1.1 Sistema de Memória Compartilhada Distribuída

Vários sistemas de memória compartilhada distribuída baseada em software já foram propostos [52, 47, 69, 51]. No entanto, DSMIO é o primeiro desses sistemas que estende o seu gerenciamento de memória principal aos dados armazenados em disco.

Nos útimos anos muitos trabalhos têm sido desenvolvidos em sistemas de *software DSM.* Porém poucos trabalhos têm se direcionado ao estudo do desempenho da memória com aplicações de banco de dados em sistemas multiprocessadores [81]. Mais especificamente, muito pouco trabalho tem sido realizado com relação a executar em *software DSM* aplicações que necessitem de uma funcionalidade intensa do sistema operacional, tal como sistemas gerentes de banco de dados paralelos em *software DSM.*

O trabalho mais próximo consta de uma tese recente e ainda não publicada [70]. Este trabalho se preocupou em categorizar os problemas encontrados ao se adaptar o *software DSM Treadmarks* para uma classe de aplicações de E/S intensiva, no caso o SGBD *PostGres* e em apresentar soluções necessárias ao código da aplicação para tornar o porte da aplicação o mais simples possível. Este trabalho não teve como preocupação obter um bom desempenho através da combinação do sistema SGBD com o sistema de *software DSM.* Nosso foco está em investigar todas as possibilidades

de melhoria no desempenho através da combinação do sistema de *software DSM* com um sistema SGBD, independente da necessidade ou não de modificações no código fonte das aplicações dos dois sistemas.

Feeley et al [82] discutem o desenvolvimento de um sistema DSM transacional. Eles utilizam o sistema CMU's Recoverable Virtual Memory [83], que fornece suporte para o armazenamento de memória persistente em discos de *log* para uma eventual recuperação do sistema. Eles distribuem variáveis de *lock* e executam operações de difusão de atualizações de *logs* para todas as máquinas do sistema de forma a fornecer coêrencia e criar um *software DSM.* O trabalho deles é similar no fato de que eles podem ser vistos como operando em uma *cache* local que pode ser acessada diretamente por qualquer processo no espaço da memória virtual. Entretanto, eles não fornecem funcionalidade completa a base de dados.

Recentemente, o trabalho de M.Costa et al [84] apresenta um algoritmo de recuperação de dados implementando um sistema de *log* para um sistema DSM com modelo LRC. O algoritmo tolera a falhas de um nó no sistema, para isto ele mantém um sistema de *log* distribuído contendo as dependências entre os dados voláteis dos processos do sistema. O algoritmo é implementado em *Treadmarks* mas a sua análise de desempenho é desenvolvida para aplicações científicas.

Seguindo esta mesma linha, vários trabalhos têm sido desenvolvidos de forma a prover tolerância falhas com sistemas de *log* e algoritmos de recuperação a falhas em sistemas de *software DSM.* Estes trabalhos, entretanto, também têm sido desenvolvidos visando aplicações científicas [85] [86] [87] [88].

O trabalho desenvolvido por Scale e Gharachorloo [50] lida com várias questões similares ao nosso trabalho mas com uma perspectiva diferente. Isto é devido principalmente às diferenças estruturais entre os *software DSM Treadmarks e Shasta.* Shasta implementa uma consistência de memória de granularidade fina, no nível do bloco da *cache*, utilizando protocolo de invalidação baseado em diretório [49]. O código binário do *hardware* do multiprocessador foi re-escrito, de forma a colocar operações de *miss check* em cada referência de memória. Shasta atende ao modelo de memória do ALPHA e devido a isto, usa a arquitetura de baixa latência do *Memory Channel* para enviar mensagens sem ter que realizar chamadas ao sistema operacional. Shasta possui suporte para utilizar de forma eficiente o *hardware* da memória compartilhada dos processadores nas máquinas individuais SMP.

O objetivo do trabalho deles também foi executar um sistema paralelo de banco

de dados em um *cluster* com um *software DSM*. Eles implementaram o sistema de banco de dados comercial *Oracle* sem alterar o seu código fonte e executaram operações de consultas do *benchmark TPC-D*.

10.1.2 Sistemas de Memória Global

Sistemas de memória global são sistemas que fornecem ao uma visão global da memória primária ao nível da aplicação.

Pela perspectiva da aplicação, o nosso trabalho apresenta similaridades fundamentais com trabalhos desenvolvidos na Universidade de Wisconsin-Madison para fornecer suporte de sistemas de memória global para sistemas SGBD. Antes de analisarmos as similaridades é importante ressaltar que os trabalhos desenvolvidos foram executados em sistemas simulados, de forma diferente do nosso trabalho, que foi desenvolvido integralmente em sistemas reais.

Originalmente os trabalhos da Universidade de Wisconsin consideram o problema de lidar com *caches* em sistemas cliente-servidor sem considerar as políticas de decisão de substituição de *cache* baseada no conteúdo de outras *caches* [36].

Franklin et al [7] lidam com o gerenciamento em sistemas cliente-servidor, onde os clientes podem acessar páginas uns dos outros como um terceiro nível na hierarquia de memória, depois de acessar a memória local e a memória do servidor. A simulação feita por eles executa um compartilhamento de página assim como no nosso trabalho, mas devido ao fato de eles se prenderem à arquitetura do SGBD, eles mantém em *cache* as variáveis de *lock* e os dados. Em um trabalho anterior [36], eles observaram que o desempenho era alterado ao se adotar esquemas simplificados que reduziam a replicação de cópias de dados entre o cliente e o servidor ou entre clientes individuais. Desta forma eles examinaram esquemas de memória globais que pudessem vir a diminuir as replicações dos dados, com o objetivo de gerar um aumento de desempenho. Estes esquemas são baseados no conceito de minimização da replicação dos dados, o que é bastante diferente da arquitetura do sistema *TreadMarks*, que é projetado para fornecer ganho de desempenho para aplicações que necessitam de acesso rápido a pequenas quantidades de dados compartilhados e não uma gerência de substituição de dados globalizada de forma a otimizar o acesso a uma grande massa de dados. O sistema *TreadMarks*, baseado no modelo LRC, tem como objetivo apresentar uma política de coerência de *cache* cuja característica fundamental é a diminuição da carga na rede, enquanto que o sistema global, se

preocupa com a diminuição das cópias no sistema e acaba por gerar um aumento exagerado de carga na rede, conseqüentemente um desempenho desastroso.

Venkartaman [80], desenvolveu um modelo simulado de um sistema multi-servidor onde a *cache* de cada servidor é gerenciada por uma política de gerenciamento de memória global. Esta política reduz a duplicação dos dados ao mesmo tempo em que utiliza a memória ociosa e que diminui a contenc cão na rede, que era o fator mais problemático no desempenho do trabalho anterior. Eles também investigaram a utilização dos discos locais dos clientes como um nível a mais na hierarquia de memória do sistema. Mas assim como no trabalho anterior, a preocupação deste trabalho está em desenvolver uma política de substituição de *cache* globalizada de forma a se otimizar o acesso a uma grande massa de dados.

10.1.3 Algoritmo de coerência de cache de um Sistema Cliente-Servidor

Comparamos o desempenho do algoritmo de consistência de *cache* do sistema cliente-servidor GOA-CBL com o sistema cliente-servidor GOA-DSMIO para diferentes cargas de trabalho assim como diferentes configurações de sistema. Adotamos uma configuração de estudo de desempenho similar a vários estudos recentes de desempenho de consistência de cache [11] [12] [35]. A diferença fundamental entre os trabalhos acima relacionados e o estudo realizado, está no fato de que desenvolvemos e implementamos algoritmos reais em cima de um Gerente de Banco de Dados real, o GOA [15], enquanto que os trabalhos relacionados foram todos desenvolvidos em simuladores. Apesar desta importante diferença, os resultados obtidos pelo sistema GOA-CBL com as cargas de teste foram bastante próximos dos resultados simulados, como pode ser visto no capítulo 7.

Mostramos com os testes realizados no capítulo 8 que o algoritmo de consistência de *cache* do sistema cliente-servidor GOA+DSMIO é um algoritmo que possui desempenho superior ao algoritmo CBL [8], algoritmo mais adotado nos dias de hoje. Isto se deve ao fato de que o sistema DSMIO permite uma redução drástica no número de mensagens necessárias para se manter a coerência dos dados compartilhados.

10.1.4 Cacheamento Cooperativo

O cacheamento cooperativo [75] é implementado por vários sistemas, tais como SLD e GMS [76]. A diferença fundamental entre estes sistemas e o sistema DSMIO, está no fato de que DSMIO implementa uma cache cooperativa de *diffs* o que caracteriza um maior aproveitamento do espaço de armazenamento dos dados modificados. Além disso, DSMIO só aproveita o cacheamento cooperativo em operações de leitura de dados; operações de escrita a dados são sempre realizadas localmente, e armazenadas sob a forma de estruturas de *diffs*.

10.1.5 Sistemas de otimização de escritas em disco

Existem ainda trabalhos que propõem uma redução na quantidade de acessos ao disco através do acúmulo em cache dos pedidos de escrita ao disco, como no caso dos sistemas LFS [65], Zebra [66] e Enwrich [77], de forma a transformá-los em um único, ou em poucos acessos otimizados ao disco. Estes sistemas otimizam a E/S de forma ortogonal a otimização realizada pelo sistema DSMIO. De forma que eles podem ser somados ao sistema DSMIO. No sistema DSMIO, o objetivo está em diminuir os pedidos de escrita em disco. Os pedidos de escrita em disco são transformados em armazenamento de *diffs* na memória principal; e os dados são recuperados na sua forma integral apenas no momento em que ocorre um *overflow de cache* ou no término da execução do programa. Nesta etapa, os sistemas acima mencionados podem ser utilizados para que os dados venham a ser escritos no disco de forma otimizada.

10.1.6 Sistemas de Arquivos Distribuídos

Assim como os sistemas de memória compartilhada distribuída, todos os sistemas de arquivos distribuídos implementam alguma forma de cacheamento de dados. As caches nesses sistemas têm sido implementadas de duas formas principais: na memória principal (como no sistema SLD [71]) ou nos discos locais (como no sistema AFS [72]). Entretanto, diversas unidades de coerência têm sido utilizadas nesses sistemas. Alguns sistemas, como o AFS, utilizam o arquivo como unidade de coerência,enquanto que outros utilizam blocos do arquivos, como Zebra [66], ou blocos do disco como o SLD [71]. O sistema DSMIO implementa as suas *caches* em memória principal e utiliza a página como unidade de coerência. No entanto,

a principal diferença entre o nosso sistema e propostas anteriores em termos de *caching* de dados de disco é que em DSMIO apenas as modificações (*diffs*) realizadas por um processador são armazenadas localmente e a uma cópia completa dos dados de disco é armazenada pelo sistema no nó *home.*

Em termos do modelo de consistência dos dados de disco, DSMIO também se diferencia bastante de sistemas de arquivos distribuídos anteriores. DSMIO adota o modelo *Lazy Release Consistency* [1], enquanto que a quase totalidade dos outros sistemas adota o modelo seqüencial de consistência [73, 66] e os mais recentes adotam o modelo *Release Consistency* [74] e *Entry-Consistency* [71].

Existe uma série de trabalhos em sistemas de arquivos paralelos e distribuídos que, assim como DSMIO, propõem otimizações nas operações de E/S ao disco. Em muitos desses trabalhos [91, 92, 93, 94] são propostas novas interfaces para o acesso à disco, permitindo assim que o usuário dirija a otimização das transferências de dados. O sistema DSMIO otimiza as operações de E/S de forma distinta desses sistemas, uma vez que DSMIO não se preocupa em fazer com que a transferência de dados de/para o disco seja mais eficiente, mas sim em diminuir a quantidade de acessos ao disco. Na verdade, assim como nos sistemas de otimização de escritas em disco, seção 10.1.5, as técnicas propostas por DSMIO são ortogonais a estas técnicas de otimização da transferência de dados, sendo também um beneficiário dessas evoluções.

Capítulo 11

Conclusões e Trabalhos Futuros

11.1 Conclusões

Aplicações paralelas de diversas áreas necessitam de sistemas de E/S com alto desempenho. Esta tese propõe e avalia os mecanismos encontrados em Sistemas de Memória Compartilhada distribuída implementados em *software* para otimizar o *caching* de dados do disco, e conseqüentemente o sistema de E/S paralela. Construímos o sistema DSMIO para um multicomputador IBM-SP e o avaliamos no contexto de diversas arquiteturas de SGBD executando aplicações de suporte à decisão.

Nossos resultados demostram experimentalmente que, ao trazer a coerência dos dados armazenados em disco ao nível da memória principal empregando os mecanismos do *software DSM*, aumenta significativamente o desempenho do sistema de Entrada/Saída.

Mostramos que trazer a coerência dos dados armazenados em disco mantida segundo um modelo de consistência seqüencial (GOA+NFS), para o nível da memória principal, com o modelo de consistência LRC (GOA+DSMIO) gera ganhos substanciais. No contexto dos nossos experimentos, obtivemos reduções de até 99% no tempo de execução da traversal T3 com atualizações do *benchmark OO7*. Este resultado confirma experimentalmente o resultado de estudos anteriores sobre desempenho entre algoritmos LRC *versus* algoritmos RC.

O modelo LRC reduz drasticamente o número de mensagens necessárias para se manter a coerência de *caches* compartilhadas. Nos nossos experimentos obtivemos reduções de até 86% com relação a um modelo RC.Mostramos, no contexto dos testes realizados, que o mecanismo de *diffs* não só permite a ocorrência de múltiplos escritores concorrentes em uma mesma página compartilhada, como também permite

uma drástica redução no tamanho das mensagens de coerência enviadas pelo sistema de E/S, em até 99% e uma redução significativa no tamanho da área de memória local necessária para se armazenar os dados a serem escritos no disco.

Avaliamos o sistema DSMIO em um ambiente SGBDOO cliente-servidor GOA+DSMIO para vários tipos de cargas. Nossos resultados demonstram experimentalmente que o *overhead* introduzido por DSMIO é insignificante. Além disso, no contexto dos testes desenvolvidos, DSMIO apresenta um algoritmo de consistência de cache muito mais eficiente que o algoritmo de consistência de cache mais adotado para os sistemas SGBD Orientado a Objeto (CBL).

Similarmente avaliamos o desempenho do sistema DSMIO no desenvolvimento de um sistema SGBDOO paralelo cliente-servidor. Demonstramos, no contexto dos testes realizados, que o aumento no desempenho devido ao emprego do sistema DSMIO, torna-o eficiente e uma opção de desenvolvimento de SGBDOO paralelo cliente-servidor capaz de unir um ambiente de programação amigável à uma arquitetura SGBDOO paralela de memória distribuída. O que possibilita o emprego de um sistema SGBDOO paralelo em sistemas escaláveis de baixo custo tais como os *cluster* de estações *beowoulf*.

11.2 Trabalhos Futuros

Pretendemos realizar experimentos com bases maiores e com um maior número de processadores para avaliar questões de *overflow* de *cache* e latência de acesso ao disco (principalmente nos casos de sobrecarga do *buffer* de E/S do sistema operacional).

Desejamos também avaliar a eficiência do sistema DSMIO com uma arquitetura SGBD escalável, com um maior número de processadores e uma base fragmentada entre os discos locais dos processadores do sistema, minimizando o compartilhamento dos recursos. Esperamos obter *overheads* de escrita a dados compartilhados desprezíveis com o sistema DSMIO para esse tipo de arquitetura.

Para que o sistema DSMIO venha a se tornar uma opção viável para a implantação de um SGBD paralelo necessitamos prover ao sistema suporte a tolerância a falhas.

Para o futuro, necessitamos também pesquisar outras classes de aplicações que venham a se beneficiar do sistema DSMIO, que são aplicações que necessitam executar atualizações em dados compartilhados em sistemas de multiprocessadores, tais

como aplicações de *On Line Transaction Processing* e aplicações de Comércio Eletrônico.

Bibliografia

[1] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, 1992.*

[2] L.Meyer, M. Mattoso. Paralelismo em SGBDOO com Memória Distribuída: uma análise do desempenho do ParGOA-MD. In *Anais XII Simpósio Brasileiro de Banco de Dados, SBC, Fortaleza, Oct.1997, pp.272-286.*

[3] F.Tavares, J. Soares, M. Mattoso. PARGOA2: Paralelismo no Servidor de Objetos Persistentes GOA. In *Relatório Técnico COPPE/UFRJ ES-367/96-Janeiro.*

[4] M.J.Carey, D.J.DeWitt and J.F.Naughton. The 007 Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data,pp.12-21,1993.*

[5] P. O'Neil. Principles of Programming Performance. In *Morgan Kaufman Publisher, 1994*

[6] M.T.Ozsu, P. Valduriez. Principles of Distributed Database. In *Prentice-Hall International Edition, 2nd Edition, 1999.*

[7] M. J. Carey, J. Franklin, M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *Computer Sciences Technical Report 1094 June 1992- Computer Sciences Department University of Wisconsin-Madison.*

[8] M. J. Franklin, M. J. Carey, M. Livny Transactional Client-Server Cache Consistency: Alternatives and Performance. In *Proceedings of the ACM Transactions On DataBase Systems, sept.1997.*

[9] S.Dar, M. Franklin , B.Jonson, D.Srivastava,M.Tan. Semantic Data Caching and Replacement In *Proceedings of the 22nd Very Large DataBases Conference,1996*

[10] M. J. Franklin, M. J. Carey. Client-Server Caching Revisited In *Computer Sciences Technical Report 1089, May 1992- Computer sciences Department-University of Wisconsin-Madison.*

[11] M. Ozsu, K. Voruganti, R. Unrau. An Asynchronous Avoidance-Based Cache Consistency Algorithm for Client Caching DBMSs. In *Proceedings of the 24th Very Large DataBases Conference,1998.*

[12] M. Carey, M. Franklin, M. Zaharioudakis. Fine-Grained Sharing in a Page server OODBMS. In *Proceedings ACM SIGMOD Conference, 1990.*

[13] M.Carey. *Private Communication, 1997.*

[14] R. G. G. Cattel. Object Data Management. In *Addison-Wesley Publishing Company, Revised Edition, 1994.*

[15] M. L. Q .Mattoso. Aspectos de Paralelismo na Gerência de Dados e Objetos no GEOTABA. In *Tese de D. Sc, COPPE/UFRJ, RJ Brasil,1993.*

[16] D.Dewitt, J. Gray. Parallel Database System: The Future of Database Processing or a Passing Fad? In *SIGMOD Record, vol.19, n.4, 1990.*

[17] D. Dewitt, J.Gray. Parallel Database Systems: The Future of High Performance Database System. In *Communications of the ACM, n.6 1992.*

[18] M. Ozsu, P. Valduriez. Distributed and Parallel Database System. In *Handbook of Computer Science and Enginneering CRC Pres, A. Tuker, 1996.*

[19] P. Valduriez. Parallel Database System: the case for shared-something. In *Proceedings of 9th International Conference on Data Engineering, 1993.*

[20] D. Dewitt and R. Gerber Multi processor hash-based join algorithms.. In *Proceedings of the 11th International Conference on Very Large DataBases, Sweden, August 1985.*

[21] L. A. Vivacqua C. Meyer. Paralelismo em SGBDOO com memória distribuída: uma implementação no PARGOA. In *Tese de Msc, COPPE/UFRJ,1997.*

[22] O2 TECHNOLOGY *The O2 System Admnistration Guide, Version 4.5, 1995.*

[23] GOA. Um servidor de Objetos Persistentes In *URL://www.cos.ufrj.br/goa,1999.*

[24] J. Archibald, J. Baer. Cache coherence protocols: Evaluating using a multiprocessor simulation model. In *ACM transactions on Computer Systems vol.4 , 1986.*

[25] K. Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings 17th International Symposium on Computer Architecture, June 1990.*

[26] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. In *IEEE Transactions on Computers, c28 , 1979.*

[27] J. R. Goodman. Using cache Memory to reduce processor-memory traffic. In *Proceedings 10th ACM Symposium on Computer Architecture, 1983.*

[28] A. Agarawel, A. Simoni, R. Hennessy, M. Horowitz. An Evolution of Directory Schemes for Caches Coherence In *Proceedings 15th International Symposium on Computer Architecture, 1988.*

[29] P. Keleher, A. Cox, W. Zwaenepoel. Lazy Release Consistency for Software Distrubuted Memory. In *Proceedings 19th annual International Symposium on Computer Architecture, 1992.*

[30] M. Carey, M. Livny. Conflict Detection Tradeoffs for Replicated data. In *Proceedings of the ACM Transactions on Database Systems, 1991.*

[31] A. Adya et al. Efficient Optmistic Concurrency Control using Loosely Synchronized Clocks In *Proceedings of the ACM SIGMOD Conference on Management of Data, 1995.*

[32] L. Lamb et al. The ObjectStore Database System In *Communications of the ACM, vol.34, 1991*

[33] O. Deux et al. The O2 system. In *Communication of the ACM vol 34, 1991*

[34] Exodus Project Group Exodus Storage manager Architectural Overview. In *Exodus Project Document, Unversity of Wisconsin, 1991.*

[35] K. Voruganti M. Ozsu R. Usurau. An Adaptative Hybrid server Architecture for Client-Caching Object DBMS In *Proceedings of the 25th Very Large DataBases Conference, 1999.*

[36] M.Carey. M. Franklin, M. Livny. E. Shekita. Data Caching Treadeoffs in Client-Server DBMS Architecture *Computer Science Technical Report 994, Univiversity of Wisconsin, 1991.*

[37] K.Li, P.Hudak. Memory coherence in shared virtual memory systems. In *ACM Transactions on Computer systems 7,4(Nov.),1989.*

[38] S. Adve, A. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proceedings of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2), February 1996.*

[39] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer, December 1996.*

[40] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symp. on Operating Systems Principles, October 1991.*

[41] M. Dubois, C. Scheurich, and F. A. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th An. International Symposium on Computer Architecture (ISCA'86), June 1986.*

[42] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessor. In *Proceedings of the 17th An. International Symposium on Computer Architecture (ISCA'90), May 1990.*

[43] J. R. Goodman. Cache Consistency and Sequential Consistency. In *Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.*

[44] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2), February 1996.*

[45] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Computer, September 1991.*

[46] D. E. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th An. International Symposium on Computer Architecture (ISCA'90), May 1990.*

[47] L. R. Monnerat and R. Bianchini. ADSM: A Hybrid DSM Protocol that Efficiently Adapts to Sharing Patterns. In *Technical Report ES-425/97, COPPE/UFRJ, March 1997.*

[48] L. R. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proceedings of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4), February 1998.*

[49] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), October 1996.*

[50] D.Scales and K. Gharachorloo. Toward transparent and efficient software distributed shared memory. in *Proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.*

[51] C. B. Seidel, R. Bianchini, and C. L. Amorim. The Affinity Entry Consistency Protocol. In *Proceedings of the 1997 International Conference on Parallel Processing (ICPP'97), August 1997.*

[52] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference, January 1994.*

[53] M. Swanson, L. Stroller, and J. B. Carter. Making Distributed Shared Memory Simple, Yet Efficient. In *Proceedings of the 3rd International Workshop on High-Level Parallel Programming Models and Supportive Environments, March 1998.*

[54] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems. In *Proceedings of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96), October 1996.*

[55] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Int'l Computer Conference(COMPCON Spring'93), February 1993.*

[56] R. Bianchini, L. Kontothanassis, R. Pinto, M de Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7h International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996.*

[57] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, October 1994.*

[58] B. N. Bershad and M. J. Zekaukas. The Midway Distributed Shared Memory System. In *Proceedings of the 93th CompCon Conference, February 1993.*

[59] C. B. Seidel. A Técnica *Lock Acquirer Prediction* e sua aplicação em Sistemas de Memória Compartilhada Distribuída. In *Tese de D. Scs, COPPE/UFRJ, RJ Brasil,1998*

[60] M.C. S. de Castro Técnicas para Deteção e Exploração de Padrões de Compartilhamento em Sistemas de Memória Compartilhada Distribuída. In *Tese de D. Sc, COPPE/UFRJ, RJ Brasil,1998*

[61] M.F. Khan, R. Paul, I. Ahmed and A. Ghafoor. Intensive Data Management in Parallel Systems: A Survey. In *Distributed and Parallel Databases International Journal, October 1999*

[62] D. Lenoski, J.Laudon, K. Gharachorlo, A. Gupta, and J. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor In *Proceedings of the 7th Annual International Symposium on Computer Architecture, pages 148-159, May 1990.*

[63] D.Womble, D. Greenberg. Parallel I/O: An Introduction . In *Proceedings of the Parallel Computing, 23(1997) 403-417.*

[64] W. E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1997 USENIX Windows/NT Workshop, pages 95 – 106, August 1997.*

[65] Mendel Rosemblum and John Outerhoout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th Symposium on Operating Systems Principles, pages 1- 15, 1991.*

[66] John Hartman and John Outerhout. The Zebra Striped Network File System. In *Proceedings od the 14th Symposium on Operating Systems Principles, pages 29-43 ,1993.*

[67] Ken Shirriff and John Outerhout. Sawmill: a high-bandwidth logging file system. In *Proceedings of the 1994 Summer USENIX Technical Conference, pages 49-60, 1994 .*

[68] Thomas E. Anderson, Michael D.Dahlin, Jeanna M.Neefe, David A. Patterson, Drew S.Roselli, and Randolph Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th Symposium on Operating System Principles, pages 109 -126,1995.*

[69] Y. Zhou, L. Iftode and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation, 1996.*

[70] T. Parker I/O-Oriented Applications on a Software Distributed-Shared Memory System. Master of Science Thesis - Computer Science Dept- Rice University,1999.

[71] R. Shillner and E. W.Felten. Simplifying Distributed File Systems Using Shared Logical Disk. In *Proceedings of the Intel Supercomputer User's Group Conference, 1995.*

[72] J. Howard, M. Kazar, S. Meness, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. In *Proceedings of the ACM Transactions on Computer Systems, vol.6, 1988.*

[73] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer, 1988.*

[74] G. Gibson, D. Stodolsky, F. Chang, W. CourtrightII, C. Demetriu, E. Ginting, M. Holland, Q. Ma, L. Neal, R. Patterson, J.Su, R.Youssef, and J.Zelenka. The Scotch Parallel Storage Systems. In *Proceedings of the IEEE CompCon conference, 1995.*

[75] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation, 1994.*

[76] M. Feely, W. Morgan, F. Poghin, A. Karlin, H. Levy, and C. Thekath. Implementing Global Memory Management in a Workstation Cluster. In *Proc of the 15th Symposium on Operationg Systems Principles, 1995.*

[77] A. Purrakayastha, C. Ellis, and D. Kotz. ENWRICH: A Compute-Processor Write Caching Scheme for Parallel File Systems. In *Proceedings of the 4th Workshop on I/O in Parallel and Distributed Systems, 1996.*

[78] A. Purrakayastha, C. Ellis, D. Kotz,N. Nieuwejaar, and M. Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the International Parallel Processing Symposium, 1995.*

[79] L. Iftode, J .P. Sigh, and K.li Scope Consistency: A Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the ACM Symposium on Parallel Algorithmis and Architectures, June 1996.*

[80] S. Venkartarman. Global Memory management for Multi-Server DataBase Systems. In *Phd thesis, University of Wisconsin, Madison, December 1996.*

[81] P. Trancoso, J. Larriba-Pey, Z. Zhang, J. Torrellas. The Memory Performance of DSS Comercial Workload in Shared-Memroy Multiprocesors In*Proceedings*

of the Third International Symposium on High Performance Computer Architecture, February 1997.

[82] M. J. Feeley, J. S .Chase, V. R. Narasayya, H. Levy. Integrating coherence and recoverability in distributed systems In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation, November 1994*

[83] M. Satyanarayanan, H. Mashburn, P. Kumas, D. Steere, J.Kistler. Lightweight recoverable virtual memory In *Proceedings of the ACM Transactions on Computer Systems, February 1994*

[84] M.Costa, P. Guedes, M. Sequeira ,N. Neves, M. Castro Lightweight Logging for Lazy Release Consistent Distributed Shared Memory In *Proceedings of USENIX Second Symposium on Operating System Design and Implementation, OSDI, 1996.*

[85] A.Kongmunvattana, N.Tzeng Lazy Logging and Prefetch-Based Crash Recovery in Software Distributed Shared Memory Systems. In *Proceedings of the 13th International Parallel Processing Symposium (IPPS'99), 1999.*

[86] O. Theel, B. Fleisch A Dynamic Coherence Protocol for Distributed Shared Memory Enforcing High Data Availability at Low Costs. In *Proceedings of the IEE Transactions on Parallel and Distributed Systems, vol7, September 1996*

[87] A.Kongmunvattana and N.Tzeng. Coherence-Centric Logging and Recovery for Home-Based Softare Dsitributed Shared Memory In *Proceedings of the IEEE, 1999*

[88] C.Morin and I.Puaut. A Survey of Recoverable Distributed Shared Virtual Memory Systems. In *IEEE Transactions on Parallel and Distributed Systems, September 1997.*

[89] C .Osthoff, R. Bianchini, M. Mattoso,C. Seidel, C. Amorim. Explorando Conceitos e Mecanismos de Memória Compartilhada Distribuída em E/S paralela In *Relatório de Pesquisa e Desenvolvimento-LNCC-05/99, Fevereiro 1999.*

[90] C. Osthoff, R .Bianchini, M .Mattoso, C. Seidel, C. Amorim. Explorando Conceitos e Mecanismos de Memória Compartilhada Distribuída em E/S paralela

In *Proceedings of XV Brazilian Symposium on Computer Architecture, October 1999*

[91] Corbett, S. Johnson, D. Feitelson. Overwiew of the Vesta Parallel File System. In *Proceedings of the ACM Computer Architecture News, 21, 1993, pp-7-15.*

[92] A.Choudhary, R. Bordawekar, S. More, K.Sivaram, R.Thakur PASSION runtime library for the Intel Paragon In *Proceedings of the Intel Supercomputer User's Group Conference, 1995*

[93] J.M. del Rosario, R. Bordawekar, A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the Computer Architecture News 21, 1993, pp-56-70.*

[94] D.Kotz/T.Cai Disk-direct I/O for MIMD multiprocessors In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI), November 1994.*

[95] URL http://www.oac.ucla.edu/rts/clustering/default.htm

[96] D. Barry, T. Stanienda Solving the Java Object Storage Problem In *Proccedings of tje IEEE Computer, vol.31, N0.11, November 1998.*