UM MECANISMO MODULAR E EFICIENTE PARA COMPARTILHAMENTO DE MEMÓRIA EM CLUSTERS

Thobias Salazar Trevisan

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 MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Vítor Santos Costa, Ph.D.

Prof. Alberto Ferreira De Souza, Ph.D.

Prof^{a}. Inês de Castro Dutra, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

JUNHO DE 2003

TREVISAN, THOBIAS SALAZAR

Um Mecanismo Modular e Eficiente para Compartilhamento de Memória em Clusters [Rio de Janeiro] 2003

X, 42 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2003)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Computação de Alto Desempenho

2 - Sistemas Operacionais

3 - Memória Compartilhada Distribuída

I. COPPE/UFRJ II. Título (série)

“If you have an apple and I have an apple and we exchange apples then you and I will still each have one apple. But if you have an idea and I have one idea and we exchange these ideas, then each of us will have two ideas.”

George Bernard Shaw

Agradecimentos

Gostaria inicialmente de agradecer à minha família pelo apoio e incentivo dado durante todo o curso. Obrigado aos meus pais, Hamilton e Aparecida, e aos meus irmãos Germano, Lucas e Jerusa, os quais formam a verdadeira base da minha vida.

Agradeço ao meu orientador prof. Claudio Amorim pelo incentivo e pela infraestrutura proporcionada para a realização desta tese. Ao meu outro orientador prof. Vítor Santos Costa pelo apoio e ajuda no estudo do kernel do Linux.

Aos meus colegas da UCPel Leonardo e Rodrigo (drigator) por terem encarado esta “viagem” de vir estudar na UFRJ.

Ao Lauro, amigo que fiz durante o período que estive no LCP. Laurinho, não vou esquecer das nossas “intermináveis” conversas e discussões sobre os mais variados temas.

E por último, a um grande amigo e sócio :-) que fiz durante o mestrado. Valeu Silvano, nunca esquecerei os momentos de diversão no LCP jogando em rede DOOM, Wolfenstein e Enemy Territory. Treina que um dia você me ganha &:)

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UM MECANISMO MODULAR E EFICIENTE PARA COMPARTILHAMENTO DE MEMÓRIA EM CLUSTERS

Thobias Salazar Trevisan

Junho/2003

Orientadores: Claudio Luis de Amorim
Vítor Santos Costa

Programa: Engenharia de Sistemas e Computação

Nesta tese propomos o sistema MOMEMTO (*MOre MEMory Than Others*), um novo conjunto de mecanismos em kernel para suportar o compartilhamento global das memórias distribuídas dos nós do cluster, permitindo à aplicação utilizar mais memória do que a disponível em qualquer nó. MOMEMTO oferece primitivas básicas de sincronização, o que permite ao programador um maior controle sobre o uso da memória e abre espaço para otimizações feitas especificamente para cada aplicação. Além disso, com estas primitivas pode-se decidir *quando* e *quais* páginas de memória compartilhada precisam ser sincronizadas. O sistema MOMEMTO foi implementado no kernel 2.4 do Linux e comparamos seu desempenho com implementações em TreadMarks e MPI de alguns benchmarks. Nos experimentos o sistema apresentou um baixo *overhead* e melhorou substancialmente o tempo de execução das aplicações. Ainda mais importante que os resultados obtidos nos experimentos é o fato de que MOMEMTO se mostra como uma alternativa promissora para o compartilhamento de memória global em clusters e abre espaço para novas pesquisas nesta área, visando soluções específicas para classes de aplicações.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A MODULAR AND EFFICIENT MECHANISM FOR MEMORY SHARING IN CLUSTERS

Thobias Salazar Trevisan

June/2003

Advisors: Claudio Luis de Amorim
Vítor Santos Costa

Department: Computing and Systems Engineering

In this thesis, we propose the MOMEMTO (MOre MEMory Than Others) system, a new set of kernel mechanisms to support global memory sharing across the physically distributed memories of cluster's nodes. A main advantage of MOMEMTO is that it allows cluster's applications to use more memory space than that is available in any single node. MOMEMTO also offers basic synchronization primitives, which allow programmer to have more control over the use of the global memory and to create opportunities for application-specific optimizations. In addition, programmers can use the synchronization primitives we propose to decide when and which shared memory's pages need to be synchronized. We implemented the MOMEMTO system in the Linux's Kernel 2.4 and compared MOMEMTO's performance against that of TreadMarks and MPI implementations of some benchmarks. Overall, our results show that MOMEMTO has low overhead and that it improves significantly the execution time of the benchmarks. Most importantly, our experimental results reveal that MOMEMTO is an promising alternative to support global memory sharing in clusters and that it opens new research avenues on exploiting MOMEMTO for application-specific solutions.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Visão Geral

Nos últimos anos tem ocorrido um significativo aumento no interesse do uso de clusters de PCs (Computadores Pessoais) como alternativa efetiva para se alcançar alto desempenho com baixo custo. Hoje em dia, clusters abrangem desde pequenos sistemas [2], utilizados em laboratórios de pesquisa e universidade por exemplo, até grandes máquinas, tais como [3, 4], utilizadas em aplicações paralelas que demandam alto poder de processamento. Clusters são utilizados tanto no ambiente acadêmico quanto no empresarial, em áreas como aplicações científicas, bancos de dados, servidores web e simulações.

Embora clusters ofereçam escalabilidade em relação ao número de CPUs, bem como espaço de memória, eles podem ser limitados pela capacidade de comunicação nó-para-nó. Deste modo, escrever aplicações paralelas eficientes que escalem pode ser uma tarefa difícil. Nós argumentamos que uma das principais dificuldades para se escrever tais aplicações é a dependência em Sistemas Operacionais (SO) tradicionais *off-the-shelf*, ou seja, disponível em qualquer loja de informática. Tradicionalmente, cada nó no cluster roda sua própria cópia do sistema operacional e alguns utilitários de comunicação de dados e compartilhamento de recursos sobre um SO de propósito geral. Em outras palavras, os SOs continuam a tratar clusters como uma coleção de computadores conectados, onde cada um trabalha de forma independente. Infelizmente, sistemas operacionais tradicionais foram projetados para um ambiente computacional diferente e não suportam todas as funcionalidades necessárias para extrair todas as vantagens que clusters oferecem. Pesquisadores muitas vezes trabalham em volta deste problema no nível da aplicação. TreadMarks [7], por exemplo, implementa um ambiente de memória compartilhada em clusters. Escrevendo drivers de comunicação que rodam nas interfaces de rede, como M-VIA [1], eliminam alguns *overheads* de comunicação mas não integram a aplicação ao

SO.

Estas soluções não são suficientes para obter o máximo desempenho de um cluster, visto que elas não preenchem o espaço entre o objetivo original dos SOs, que foram projetados para computadores pessoais e servidores, e questões específicas de desempenho que surgiram com os clusters.

Nosso trabalho concentra-se no problema de fornecer uma abstração de memória compartilhada, procurando alcançar um melhor desempenho para a aplicação. Primeiro, nós observamos que várias aplicações paralelas, de NAS-benchmarks [9] a sistemas de bancos de dados [14], freqüentemente precisam não só de escalabilidade de processadores, mas também de *escalabilidade de memória*. Por exemplo, assuma um cluster com 16 máquinas, onde cada uma tem 512 MB de memória RAM, rodando um NAS-benchmark CLASS-A (aplicações que utilizam muita memória) ou um banco de dados com uma grande quantidade de dados que requeira 8GB de memória. Infelizmente, vários sistemas software DSM (Distributed Shared Memory) atuais [29, 7, 40, 39, 23, 19] ficam limitados ao tamanho da memória local de cada nó e, assim, não podem explorar todas as vantagens das memórias agregadas dos nós para compartilhamento. Nós argumentamos que uma solução eficiente para este problema requer adaptar o gerenciamento de memória virtual do SO para o cluster.

Nossa segunda observação é que sistemas DSM provêem mecanismos de coerência sofisticados e poderosos para assegurar consistência de memória, e a implementação destes sistemas pode ser complexa e introduzir *overheads* substanciais. Entretanto, muitas aplicações requerem formas mais simples para garantir consistência de memória e podem obter melhor desempenho se usarem um mecanismo mais apropriado. Cada aplicação tem suas próprias características, comportamento, padrões de acesso, sendo que, estas informações podem levar o sistema operacional a um melhor gerenciamento de sua memória e ao compartilhamento mais eficiente dos dados entre os nós para cada aplicação.

Em casos particulares pode valer a pena o esforço de implementar mecanismos específicos para suportar aplicações populares, tais como bancos de dados, simulações e aplicações científicas.

Nesta tese nós propomos a inclusão do seguinte conjunto de abstrações no kernel para o gerenciamento de memória global em clusters:

- No nível mais baixo, nós estendemos o mecanismo de memória virtual do SO, com mínimas modificações nas estrututas do kernel, para enxergar o cluster;

- Oferecemos um mecanismo modular que provê primitivas básicas de sincronização com mínimo *overhead*.

Nosso trabalho também é inspirado por pesquisas anteriores em sistemas operacionais. Como em nano-kernels [20], nós argumentamos que os usuários devem ter a opção de como conseguir máximo desempenho, e isto pode ser alcançado oferecendo diferentes interfaces para o SO, ou seja, tornando-o mais adaptável à aplicação. O sucesso de sistemas operacionais *open-source*, como Linux e os BSDs, proporcionaram acesso a imensa documentação e a clarificação das interfaces do SO, contribuindo para que *internals* do kernel se tornassem disponíveis e compartilhadas livremente. Assim, muitos usuários se acostumaram a aplicar *patches* e a reconfigurar o kernel para as necessidades do seu sistema.

Nesta tese apresentamos o sistema MOMEMTO (*MOre MEMory Than Others*), o qual estende o kernel do Linux 2.4 incluindo um conjunto de mecanismos que suportam memória compartilhada global. No nível mais baixo, o sistema altera o mecanismo de memória virtual (*Virtual Memory* - VM) do Linux e o mecanismo de rede, estendendo-os para proverem um mecanismo básico de memória compartilhada com granulosidade de uma página. Acima desta camada, provemos um conjunto de módulos que utilizam estes mecanismos para oferecerem um conjunto de abstrações ao usuário. Resultados preliminares demonstram que o sistema MOMEMTO alcança um bom desempenho quando comparado tanto a TreadMarks[7] como a MPI [5]. Em particular, MOMEMTO pode executar aplicações que requerem mais memória que TreadMarks pode oferecer.

1.2 Contribuições da Tese

As principais contribuições da tese são:

- A proposta de um modelo em kernel para agregar a memória dos nós do cluster.
- A implementação e avaliação de um mecanismo em kernel para o compartilhamento de memória em clusters.

1.3 Organização da Tese

A tese está organizada da seguinte forma. No capítulo 2 abordamos os conceitos básicos de sistemas *software DSM*, seus problemas e soluções. No capítulo 3 descrevemos o sistema MOMEMTO e sua implementação. Os experimentos são descritos e analisados no capítulo 4. Os trabalhos relacionados são abordados no capítulo 5. Por fim, no capítulo 6 apresentamos as conclusões e sugestões para trabalhos futuros.

Capítulo 2

Conhecimentos Básicos

2.1 Sistemas DSM

Em clusters, a memória de cada nó não é compartilhada fisicamente com a dos outros, deste modo toda a comunicação entre processos é realizada sobre a rede.

Atualmente, os principais paradigmas para a programação em cluster são baseados em passagem de mensagens, utilizando protocolos como PVM[22] ou MPI[5], e em sistemas DSM (Distributed Shared Memory), implementados tanto em hardware como em software.

No paradigma baseado em passagem de mensagens, o sistema de memória distribuída é totalmente exposto ao programador e, assim, toda a comunicação entre processos é realizada explicitamente pelo mesmo. O programador precisa saber *onde* estão os dados, decidir *quais* e para *quem* eles precisam ser enviados, tornando a programação uma tarefa complexa, especialmente para aplicações com estrutura de dados complexa.

Por sua vez, no paradigma baseado em memória compartilhada distribuída (Distributed Shared Memory - DSM) é criada uma abstração de memória compartilhada, fornecendo um espaço de endereçamento compartilhado. Uma das principais vantagens deste modelo está em esconder do programador a arquitetura de memória distribuída e prover uma extensão natural do modelo de programação seqüencial. A Figura 2.1 apresenta a estrutura geral de um sistema DSM, onde o software ou hardware DSM fornece a ilusão de um espaço de endereçamento único, alterando o comportamento das memórias que estão fisicamente distribuídas.

Utilizando DSM, uma aplicação pode ser escrita como se estivesse executando em um multiprocessador de memória compartilhada, acessando os dados compartilhados com operações normais de leitura e escrita. A tarefa de passagem de mensagens é escondida do programador.

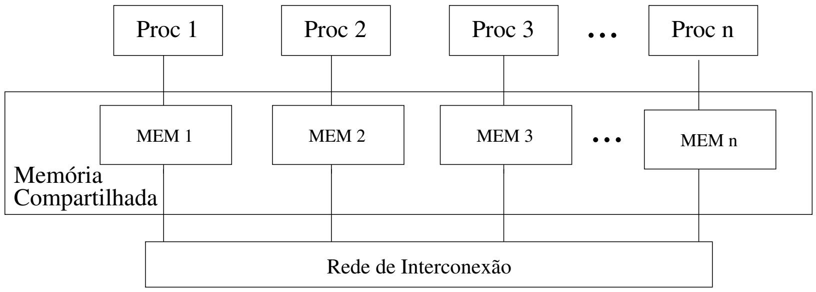

Figura 2.1: Estrutura geral de um sistema DSM

Comparando sistemas de hardware e software DSM, a altíssima relação custo/benefício impede que os sistemas hardware DSM sejam acessíveis à maioria dos usuários. Em contra partida, os sistemas de software DSM como Treadmarks [7], HLRC[40], Orca[10], CASHMERE[19], ADSM [31], JIAJIA [23], entre outros, têm sido amplamente estudados e utilizados devido à sua facilidade de implementação e ao seu baixo custo.

2.2 Desempenho de Sistemas Software DSM

Sistemas de memória compartilhada distribuída implementados em software (*software DSM*)[28] têm como objetivo oferecer uma abstração de memória compartilhada para aplicações paralelas que são executadas em arquiteturas de memória fisicamente distribuída, tais como as dos clusters.

Para a manutenção da coerência dos dados é necessário que as escritas a posições da memória compartilhada feitas por um processador sejam propagadas para os demais processadores do sistema. Esta propagação de escritas pode ser feita por protocolos de invalidação ou de atualização. No protocolo de invalidação, uma modificação em um dado compartilhado feita localmente é vista em um processador remoto através de uma mensagem de invalidação. Ao receber a mensagem, o processador remoto invalida o dado modificado, de maneira que um acesso subseqüente a este dado gera uma falha de acesso, para somente então a versão atual do dado ser buscada. No protocolo de atualização os dados não são invalidados, uma vez que a mensagem que informa que um dado foi modificado já carrega a sua nova versão, de maneira que, em um próximo acesso a este dado, não ocorrerá uma falha de acesso.

O protocolo de atualização diminui o número de falhas de acesso na memória local mas, por outro lado, ele induz um número maior de mensagens que o encontrado no protocolo de invalidação, já que muitas vezes a atualização não é vista antes que outras atualizações sejam feitas [13].

Os sistemas *software DSM* sofrem de um alto tráfego de comunicação e de *overheads* induzidos pelo protocolo de coerência que limitam seu desempenho [24]. Várias alternativas foram criadas para diminuir esse tráfego como, por exemplo, empregar modelos de consistência relaxados [27, 11, 15, 25]. Um modelo de consistência de memória especifica quando operações de coerência de dados devem tornar-se visíveis para os outros processadores, ou seja, quando as modificações feitas em dados compartilhados devem ser observadas pelos processadores. *Software DSM* baseados nestes modelos podem reduzir os *overheads* atrasando e/ou restringindo a comunicação e as transações de coerência o máximo possível. *Softwares DSM* baseados em consistência relaxada e que permitem "múltiplos-escritores" tentam reduzir a comunicação e os custos de coerência ainda mais, permitindo que dois ou mais processadores modifiquem concorrentemente suas cópias locais de dados compartilhados e combinem o resultado destas modificações nas operações de sincronização[27, 7]. Esta técnica diminui o efeito negativo do falso compartilhamento em sistemas cuja unidade de coerência é grande, como os *softwares DSM* que utilizam a página de memória virtual como unidade de coerência. O falso compartilhamento ocorre quando múltiplos escritores acessam dados não relacionados que estão localizados na mesma página e pelo menos um acesso é uma escrita ao dado. TreadMarks [7] e HLRC [40] são exemplos de sistemas que garantem a consistência de forma relaxada, utilizando o modelo *Lazy Release Consistency* (LRC) [27], e que permitem a existência de múltiplos escritores. O protocolo LRC atrasa a propagação de mensagens de coerência até pontos de sincronização como *lock* ou barreiras.

Um exemplo de software DSM que utiliza a técnica de múltiplos escritores é o TreadMarks. Neste sistema, uma página é inicialmente protegida contra escrita, de forma que na primeira escrita a ela é gerada uma violação de acesso. Durante o tratamento desta violação é feita uma cópia exata da página (um *twin*) e a escrita à cópia original da página é liberada. Quando a coleta das modificações é necessária, o *twin* e a versão corrente da página são comparadas para criar uma codificação das modificações (um *diff*). A utilização de *twins* e *diffs* em TreadMarks permite aos processadores modificarem simultaneamente suas cópias locais de páginas compartilhadas.

Desta forma, quando ocorre uma falha de acesso, o processador consulta sua lista de notificações de escrita para descobrir quais os *diffs* necessários para atualizar a página. O processador então solicita os *diffs* correspondentes e espera que eles sejam criados, enviados e recebidos. Depois de recebidos todos os *diffs* requisitados, o processador que sofreu a falha os aplica à sua cópia desatualizada. Uma descrição mais detalhada de TreadMarks pode ser encontrada em [7].

A maioria dos protocolos que permitem múltiplos escritores utilizam o mecanismo de *twinning* e *diffing*. Esses protocolos conseguem diminuir o efeito do falso compartilhamento, permitindo que vários processadores alterem concorrentemente uma página compartilhada. Porém, o suporte a múltiplos escritores cria custos adicionais no caso de páginas que não estão sujeitas ao falso compartilhamento, pois é necessário detectar, armazenar e aplicar modificações a essas páginas. Estes custos adicionais podem ser eliminados no caso de páginas que não sofrem falso compartilhamento, limitando as alterações a apenas um processador por vez. Assim, foi proposta a utilização de técnicas de adaptação entre protocolos único escritor e múltiplos escritores. Uma descrição detalhada desta técnica pode ser encontrada em *Adaptive TreadMarks* (ATmk) [8], ADSM [31, 32] e HAP[39].

Várias outras técnicas foram pesquisadas para aumentar o desempenho de *softwares DSM*. Dentre estas pode-se destacar: pré-busca [12, 26, 33] e adaptação entre protocolos único escritor e múltiplos escritores [8, 31, 32].

Na técnica de *prefetching* (pré-busca) o dado remoto é buscado com antecedência, isto é, antes de ser efetivamente necessário, tentando diminuir a latência de acesso a dados remotos de softwares DSM em tempo de execução. A utilização desta técnica de forma efetiva em softwares DSM é difícil devido a duas razões principais:

- Pode não ser fácil prever os acessos futuros a dados;
- Pré-buscas geram muitos *overheads* quando solicitadas desnecessariamente devido ao alto tempo gasto para gerar, enviar e receber estes dados que não foram necessários.

As principais fontes de atraso em *softwares DSM* estão relacionadas à latência de comunicação e às ações de coerência [34]. Latência de comunicação causa atrasos no processamento, degradando o desempenho do sistema. As ações de coerência também podem afetar negativamente o desempenho do sistema. Como o modelo de consistência de memória é a interface entre o programador e o sistema, ele especifica como a memória irá aparecer para o programador. Isto influencia na complexidade de programação e no desempenho. Um modelo de consistência de memória rígido torna a programação mais fácil, em contra-partida deixa menos oportunidades para otimizações. Além disso, diferentes tipos de aplicações requerem diferentes tipos de modelos de consistência, visto que a escolha do modelo influenciará no desempenho da aplicação.

Como vimos, existem várias técnicas para melhorar o desempenho de sistemas software DSM, por exemplo, protocolo de invalidação ou atualização, técnicas de múltiplos escritores, técnica de pré-busca, entre outros. Entretanto, estas opções não alcançam bom desempenho para todas as aplicações. Por exemplo, quando utilizado em aplicações com único escritor (*single-writer*), o suporte a múltiplos escritores pode degradar o desempenho do sistema, pois os mecanismos de *twin* e *diff* geram *overhead* e complexidade que não serão úteis para a aplicação. O problema é que os custos de desempenho desses sistemas otimizados ainda são muito altos e não conseguem melhorar o desempenho para todas as aplicações.

Capítulo 3

MOMEMTO

3.1 O Sistema

MOMEMTO foi projetado para permitir a aplicação compartilhar as memórias dos nós do cluster como uma extensão transparente da memória local. Mais especificamente, é um sistema em modo kernel que visa melhorar a utilização das memórias agregadas dos nós do cluster oferecendo à aplicação uma área de memória compartilhada maior do que a proporcionada por softwares DSM atuais [36]. No seu nível mais baixo, o sistema estende o mecanismo de memória virtual do kernel para prover um mecanismo básico de compartilhamento de páginas de memória. Como MOMEMTO utiliza a funcionalidade do kernel do Linux, começaremos apresentando o gerenciamento de memória virtual neste sistema, mas outros sistemas UNIX [37, 30] são semelhantes.

No Linux, todas as informações relacionadas com o espaço de endereçamento do processo estão incluídas na estrutura referenciada pelo campo mm no descritor do processo (*struct task_struct*). O campo mm é um ponteiro para uma estrutura do tipo mm_struct. Cada processo tem uma mm_struct.

O Linux implementa a área virtual de memória (VMA) através da estrutura vm_area_struct. Uma VMA é um intervalo contíguo de endereço linear, ou seja, uma região homogênea de memória virtual de um processo. Cada processo tem uma lista de VMAs, como ilustra a Figura 3.1. Exemplos de VMAs são:

- VMA que armazena o executável;
- VMA para variáveis globais;
- VMA para a pilha (*stack*).

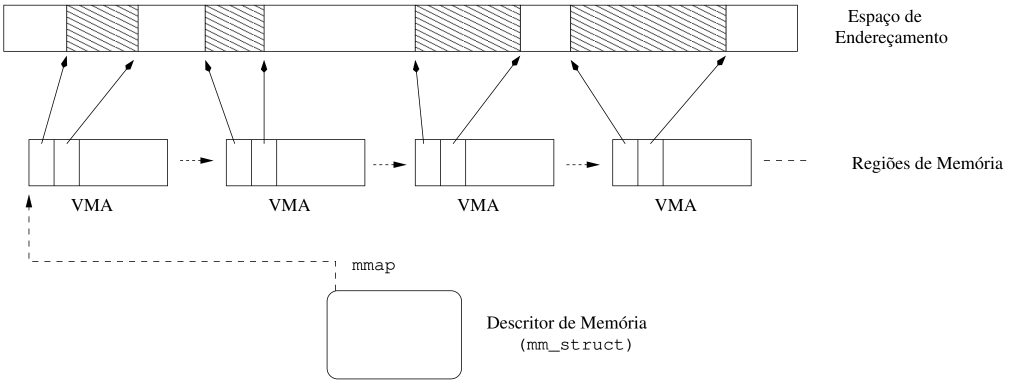

Figura 3.1: Memória Virtual no Linux

Todas as VMAs de um processo são ligadas através de uma árvore. O kernel encontra uma área de memória através do campo $mmap$ da mm_struct. Quando o kernel tem um endereço e quer encontrar a qual VMA este endereço pertence, ele usa a função $find\_vma$, que pode ser encontrada em *mm/mmap.c*. As regiões de memória de cada processo podem ser acessadas no arquivo */proc/pid/maps* onde pid é o id do processo. Alguns campos da vm_area_struct são:

- vm_start e vm_end: endereços de memória virtual de início e de fim da VMA;
- vm_mm: um ponteiro para a mm_struct;
- vm_next: ponteiro para a próxima VMA;
- vm_ops: um conjunto de funções que o kernel deve chamar para operar sobre esta área. Este campo é um ponteiro para uma estrutura do tipo $vm\_operations\_struct$.

A estrutura $vm\_operations\_struct$ manipula eventos específicos sobre uma VMA. Um dos campos desta estrutura é o $nopage$. Quando um processo tenta acessar uma página que não está na memória e pertence à uma VMA válida, ou seja, tem um endereço válido, esta função é chamada, se estiver definida, para esta VMA. Se a função $nopage$ não estiver definida para esta área o kernel aloca uma nova página. É importante salientar que um endereço pode ser coberto por uma VMA mesmo se o kernel ainda não tenha alocado uma página para armazená-lo. O kernel do Linux realiza paginação sob demanda, ou seja, uma página só será alocada quando o processo acessá-la. Informações detalhadas sobre o kernel e o sistema de memória virtual do Linux podem ser encontradas em [16, 6].

MOMEMTO cria e gerencia uma VMA do processo, deixando as demais serem gerenciadas pelo sistema operacional. Um processo usa o sistema da mesma maneira como

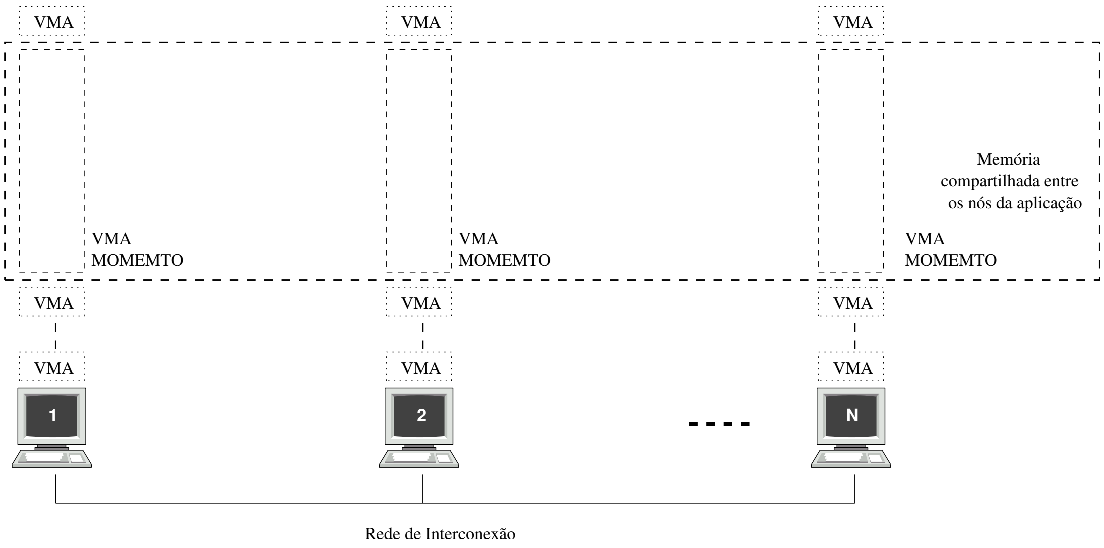

Figura 3.2: Estrutura básica do MOMEMTO - 1

se estivesse utilizando um `mmap()` para memória anônima. Mais precisamente, o usuário requisita um novo intervalo de endereços, MOMEMTO cria um novo intervalo de memória e o insere na lista de VMAs do processo. Na prática, o processo utilizará esta memória do mesmo modo como se ele tivesse feito uma chamada às funções `mmap()` ou `malloc()`.

A memória de MOMEMTO é alocada sob demanda. Em uma falha de página, o kernel chama a função `nopage` para realizar o tratamento adequado. Quando a falha de página ocorre em um endereço que pertence à VMA gerenciada por MOMEMTO, é realizado um tratamento específico para esta página. Este processo será descrito na seção 3.4.2.

A Figura 3.2 demonstra como MOMEMTO interage com uma aplicação. Cada processo tem vários segmentos de memória virtual em seu espaço de endereçamento. Os segmentos normais do Linux são locais a cada nó. Em contraste, os segmentos de MOMEMTO são compartilhados entre os nós do cluster. Do ponto de vista da aplicação, o espaço compartilhado é visto com um único bloco contíguo de memória. Do ponto de vista do sistema, os segmentos compartilhados de cada nó são agregados para formar um grande segmento compartilhado, onde cada nó pode ter mapeado em memória algumas páginas do segmento compartilhado, mas não necessariamente todas.

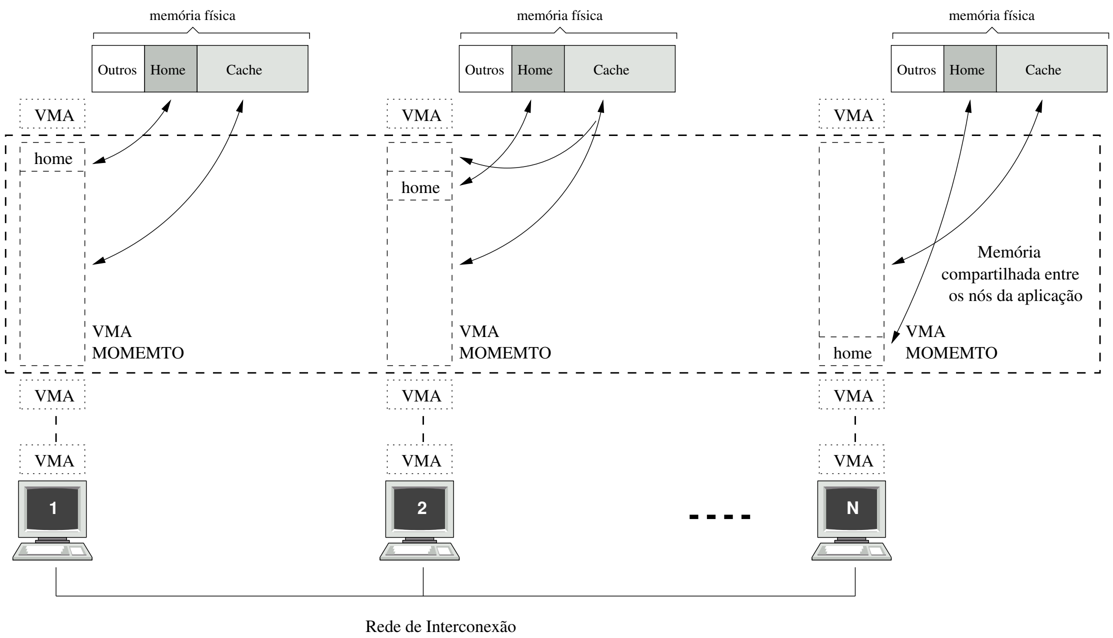

Figura 3.3: Estrutura básica do MOMEMTO - 2

3.2 Design

MOMEMTO permite uma alta flexibilidade para a aplicação. Isto é alcançado através de chamadas de sistema que controlam políticas ou através de diferentes módulos que implementam diferentes mecanismos de coerência.

No nível mais baixo, MOMEMTO somente compartilha páginas entre os nós. Para agregar a memória dos nós, utilizamos protocolo baseado em residência (*home*), onde cada nó é responsável por uma fração do número total de páginas, como ilustra a Figura 3.3. Inicialmente, cada nó tem em memória somente as páginas das quais ele é a residência. Para as demais, uma página física só será alocada quando o nó acessá-la. Quando este acesso ocorrer, o sistema buscará uma cópia desta página na sua residência.

A estratégia utilizada atualmente para decidir qual nó será a residência de uma página é a seguinte: as páginas são divididas estaticamente entre os nós e cada nó aloca as páginas que lhe pertencem na inicialização, gerando uma falha de página para cada uma.

MOMEMTO oferece primitivas básicas para o compartilhamento de dados entre os nós do cluster, ou seja, não garante consistência. O sistema não é responsável por sincronização e tampouco por atualização. Ao contrário, esta é uma tarefa para camadas mais altas e possivelmente para a aplicação. A maioria dos protocolos DSM realizam atualização de páginas em pontos de sincronização, mas a melhor política de atualização varia entre aplicações. Freqüentemente a aplicação conhece quais processadores têm cópia de uma página e pode facilmente descobrir quais páginas precisam ser atualizadas. Entretanto, mecanismos genéricos de mais alto nível podem ser implementados através de módulos de kernel ou bibliotecas para aplicações. O MOMEMTO procura, portanto, manter-se como uma ferramenta básica que pode ser utilizada por várias aplicações e permite ao programador um maior controle sobre o gerenciamento da memória compartilhada e é um mecanismo sobre o qual serviços deverão ser implementados, tais como DSM, MPI, cache para banco de dados, etc.

3.3 Interface

A interface entre MOMEMTO e a aplicação se dá através de três novas chamadas de sistema adicionadas ao sistema operacional.

mmto_mmap, *mmto_load* e *mmto_sync*.

A *mmto_mmap* faz o mapeamento de uma área de memória virtual que irá ser compartilhada entre os nós. Esta chamada de sistema, similar a função `mmap`, recebe como parâmetros um número indicando em bytes a quantidade de memória a ser alocada e uma *flag* que especifica como as residências serão distribuídas. Com esta *flag* pode-se implementar diferentes algoritmos para a distribuição das residências. Atualmente, o valor suportado pelo sistema para esta *flag* é `SEQUENTIAL`, descrito na próxima seção.

Por sua vez, a chamada de sistema *mmto_load* recebe como parâmetro um arquivo texto com os nós que irão compartilhar memória. Todos os nós devem executar esta chamada de sistema para iniciar o sistema. Usando um arquivo para especificar os nós, podemos reaproveitar arquivos gerados e utilizados por outros sistemas. Por exemplo, MOMEMTO pode ler arquivos gerados por escalonadores de tarefas como PBS[38] e SGE[35], e assim saber quais nós foram alocados para executar a aplicação.

A *mmto_sync* provê um mecanismo para a sincronização. Ela recebe como argumento o endereço de uma página virtual compartilhada e uma *flag* que determina qual ação deve ser tomada sobre esta página. Atualmente existem duas opções para esta *flag*:

- `MMTO_UPDATE`, que permite ao usuário atualizar uma página de memória compartilhada. A atualização é feita enviando o conteúdo da página para a residência, a qual então envia para os nós que tiverem cópia;

- `MMTO_FREE`, que possibilita a liberação de uma página compartilhada residente fisicamente em memória.

3.4 Protótipo

O protótipo atual de MOMEMTO foi implementado sobre o Linux kernel 2.4 e provê funcionalidade essencial para o compartilhamento de memória. Escolhemos Linux por diversas razões, principalmente porque ele é amplamente utilizado e é *open source*. Uma outra vantagem é o seu sofisticado mecanismo de módulos. O Linux utiliza *linkagem* dinâmica em kernel, o que permite carregar e descarregar módulos no kernel enquanto o mesmo está rodando. Deste modo, pode-se adicionar código especializado para uma determinada aplicação sem precisar reinicializar a máquina e sem alterar o funcionamento básico do sistema operacional para as demais aplicações. A maior parte do sistema foi implementado utilizando esta funcionalidade com mínimas mudanças no kernel. As mudanças no kernel foram necessárias apenas para tornar alguns símbolos do kernel visíveis a módulos, e foram implementados como um *patch*. Estes símbolos são: `vm_area_cachep`, `insert_vm_struct`, `make_pages_present`, `lru_cache_add`, `mark_page_accessed` e `zap_page_range`. Com a utilização de módulos para implementar o sistema, deixamos o código mais portável. Por exemplo, a maioria das distribuições Linux modificam os kernels distribuídos em suas versões, o que pode levar a erros ao aplicar um *patch* como o de MOMEMTO. Além disto, seria necessário distribuír um *patch* específico para cada versão do kernel, os quais estão sempre mudando, enquanto que, com a utilização de módulos basta compilá-los para o kernel do usuário.

3.4.1 Chamadas de Sistema

Nesta seção serão descritos os detalhes técnicos de todas as chamadas de sistemas adicionadas ao sistema operacional por MOMEMTO.

3.4.1.1 mmto_mmap

Esta chamada de sistema é utilizada para alocar um intervalo de endereços de memória virtual. As principais tarefas são:

- Todos os nós criam uma VMA com memória anônima[1] [37] para a área compartilhada. O tamanho desta área é obtido como argumento na chamada de sistema e inserida na lista de VMAs do processo.

[1] Memória anônima é a que não está vinculada a nenhum arquivo, i.e., a mesma memória utilizada em um `malloc`.

- A esta VMA é atribuída a *flag* `VM_LOCKED`, em outras palavras, todo este intervalo de endereços é marcado como não substituível (*s*wapable), ou seja, o kernel não pode retirar estas páginas da memória.

- Finalmente, é configurado o tratamento para qualquer falha de página que ocorra nesta VMA, que será descrito a seguir.

3.4.1.2 mmto_load

A função desta chamada de sistema é inicializar as estruturas internas do sistema. Tem como argumento um arquivo contendo os nós que irão compartilhar memória. Dentre suas principais atribuições pode-se citar:

- A divisão das residências, que nesta primeira implementação é feita de forma estática e por blocos. Assim, cada nó recebe $1/N$ do número total de páginas. O arquivo recebido como argumento nesta chamada de sistema contém os *hosts* que irão compartilhar memória. O primeiro nó do arquivo recebe o primeiro bloco com $(1/N)$ de memória compartilhada e assim por diante. Esta técnica foi utilizada para simplificar a implementação e para facilitar o compreendimento da distribuição das residências.

- Após esta divisão, é gerada uma falha de página para cada uma das que o nó é a residência, ou seja, as páginas que o nó é a residência estão sempre em memória. Para as demais, uma página física só será alocada quando o nó acessar a página.

- Por fim, são criadas $N - 1$ threads de kernel, onde N é o número total de nós. Cada thread de kernel é responsável por receber pedidos de falha de página vindo de outros nós para páginas que ele é a residência e também por receber atualizações de páginas.

Há algumas vantagens na utilização de threads de kernel por MOMEMTO, por exemplo, a troca de contexto (*context switch*) entre threads de kernel são mais leves do que entre processos comuns, porque elas rodam somente em modo kernel. Outra vantagem é que as utilizando o nosso sistema realiza menos trocas de contextos. Por exemplo, quando um processo recebe uma atualização de página, a thread de kernel que está conectada com o nó que enviou a atualização é acordada, faz a cópia direta para a página do usuário e voltar a dormir. Todo o processo ocorre sem a necessidade de troca de contexto.

3.4.1.3 mmto_sync

A *mmto_sync* provê um mecanismo para a sincronização. Ela recebe como argumento o endereço de uma página virtual compartilhada e uma *flag* que determina qual ação deve ser tomada sobre esta página. Atualmente as duas *flags* suportadas são:

- `MMTO_UPDATE`, que permite ao usuário decidir *quando* e *qual* página compartilhada os outros nós devem receber uma cópia atualizada. Quando o usuário atualiza uma página, se o nó for a residência da página uma cópia atualizada é enviada para quem tiver cópia, caso contrário, é enviada primeiro uma cópia para a residência da página, o qual então reenvia para quem tiver cópia.

- `MMTO_FREE`, que possibilita a liberação física de uma página compartilhada de memória, ou seja, libera uma página residente em memória. Esta opção só tem efeito em páginas que o nó não for a residência.

3.4.2 Falha de Página

Na inicialização é forçada uma falha de página para cada uma das páginas que o nó é a residência. Estas falhas de páginas são utilizadas para o nó ter em memória as quais ele é a residência. O primeiro acesso às outras páginas compartilhadas resultam em um pedido de página que é enviado à sua residência, a qual então envia uma cópia para o nó que realizou o pedido e marca um bit em uma estrutura interna de MOMEMTO, indicando que aquele nó tem uma cópia.

Os nós se comunicam utilizando sockets TCP. Para diminuir o *overhead* do TCP/IP, uma conexão é estabelecida entre os nós na inicialização, ou seja, cada nó abre uma conexão com uma thread de kernel dos nós remotos e assim todos permanecem conectados entre si. O protocolo implementado por MOMEMTO é basicamente composto de mensagens de *pedido* e *resposta* que são enviadas sobre a rede TCP/IP. Estas mensagens são utilizadas para pedir ou atualizar uma página remota. Para cada atualização realizada pela aplicação apenas 4 bytes são adicionados pelo sistema, que são utilizados para informar qual página receberá os dados atualizados. O resto dos bytes trafegados são dados úteis da aplicação e é gerada apenas uma mensagem para esta operação. Por sua vez, na falha de página são utilizadas duas mensagens: uma com 4 bytes para especificar a página requisitada e outra para receber os dados referentes à ela.

Como todas as VMAs têm o mesmo tamanho, mas podem ter sido alocadas com endereços virtuais diferentes, os nós se comunicam utilizando o deslocamento (*offset*) dentro da VMA em vez do endereço virtual da página.

3.4.3 Consumo de Memória

MOMEMTO tem um baixo consumo de memória quando comparado com as estruturas de dados de memória virtual do kernel. Todas as estruturas requeridas pelo sistema são inicializadas quando a aplicação é executada e liberadas quando ela termina.

O sistema utiliza três principais estruturas de dados que são alocadas como vetores contíguos:

- `home` é um vetor de bytes onde o índice i tem a residência para a página i, o que permite implementar mecanismos genéricos para distribuição das residências;

- `copy` é um vetor de campos de bits, um para cada página que o nó é a residência. Cada bit do campo de bits indica se o nó correspondente tem ou não uma cópia da página;

- `last` indica qual nó acessou por último a página.

Deste modo, os gastos de memória em MOMEMTO são:

- 6 bytes por página da qual o nó é a residência;

- 1 byte para as outras páginas.

MOMEMTO não modifica a estrutura VMA do kernel. As estruturas de dados requeridas por MOMEMTO são alocadas quando a aplicação começa a executar e liberadas quando a aplicação termina e além disto, são locais ao módulo. Assim, MOMEMTO não adiciona nenhum *overhead* na execução de aplicações que não o utilizam e nem para a execução do kernel.

3.4.4 Barreira

Com o objetivo de oferecer sincronização, foi implementada uma barreira centralizada como uma biblioteca para a aplicação no nível de usuário. Esta barreira somente sincroniza a computação, ou seja, não implementa consistência de memória. Quando um nó alcança uma barreira ele envia uma mensagem de 4 bytes , com o número da barreira, para o gerente de barreira [17]. O gerente espera por mensagens de todos os nós envolvidos e, após receber as mensagens de todos os nós, envia uma mensagem também de 4 bytes liberando-os da barreira. Toda a comunicação realizada na barreira é através de sockets TCP.

Capítulo 4

Experimentos

Com os experimentos realizados, espera-se avaliar o impacto do protótipo do MOMEMTO. Entretanto, salientamos que uma comparação direta de outros sistemas com MOMEMTO é difícil, visto que o sistema utiliza o paradigma de memória compartilhada mas, ao mesmo tempo, não garante consistência de memória. Para tentar realizar uma comparação substancial desta tese, comparamos MOMEMTO com um software tradicional do paradigma de memória compartilhada (TreadMarks) e um software do paradigma de troca de mensagens (MPI).

4.1 Ambiente de testes

Todos os testes foram realizados em um cluster de 8 nós, onde cada nó contém um processador Pentium III de 650MHz. Cada nó possui 512 MBytes de memória RAM, cache L2 de 256 KBytes e duas caches L1 (uma de instrução e outra de dados) com 16 KBytes cada. Os nós estão conectados por uma rede *Fast Ethernet* e cada nó tem uma interface de rede *onboard Ethernet Pro 100*. Utilizamos em nossos experimentos o sistema operacional Linux com o kernel 2.4.17, TreadMarks versão 1.0.3.2 e MPI LAM 6.5.3.

4.2 Aplicações

Para avaliar MOMEMTO escolhemos duas aplicações: multiplicação de vetores e *Integer Sort* (IS), as quais são descritas e analisadas a seguir.

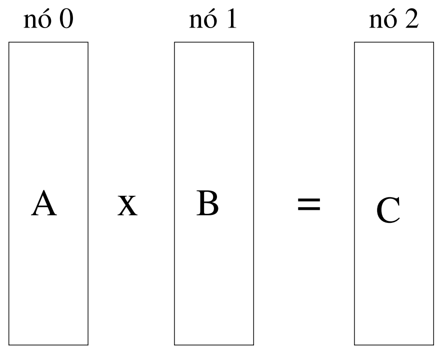

Figura 4.1: Disposição das residências

4.3 Multiplicação de Vetores

Inicialmente escolhemos uma multiplicação de vetores (MV), onde o elemento i de um vetor é multiplicado com o elemento i do outro, resultando em um vetor de mesmo tamanho. Esta aplicação foi escolhida devido ao fato de ter um padrão de acesso bem definido e de requerer uma grande quantidade de memória. A aplicação consiste de três vetores $C = A * B$, onde cada nó inicializa um vetor. As residências são atribuídas de forma estática e contígua, assim garantimos que cada nó será a residência de um dos vetores, como demonstra a Figura 4.1.

A computação do vetor C, ilustrada na Figura 4.2, é dividida igualmente entre três nós, onde cada um calcula uma terça parte do vetor resultante.

4.3.1 Resultados

Foram realizados dois experimentos. No primeiro experimento, chamado MV Pequeno, nós utilizamos 36.000 páginas compartilhadas (12.000 por vetor), o que representa um total de 141 MB de memória compartilhada. No segundo teste, chamado MV Grande, foram utilizadas 211.194 páginas compartilhadas (70.398 por vetor), representando um total de 825 MB. Ressaltamos que esta quantidade de memória é maior que a memória física disponível em qualquer nó do cluster.

A Figura 4.3 mostra o tempo total no teste MV Pequeno para MOMEMTO e TreadMarks (TMK) [7]. Ambos sistemas utilizam o mesmo algoritmo para realizar a multiplicação de vetores. Como pode-se observar, há uma melhora no tempo de execução da aplicação de 48% para MOMEMTO quando comparado à versão em TMK. Esta melhora está diretamente relacionada com o número de falhas de página, chamadas de sistema e latência de interrupção em ambos sistemas. Como o mecanismo de memória compar-

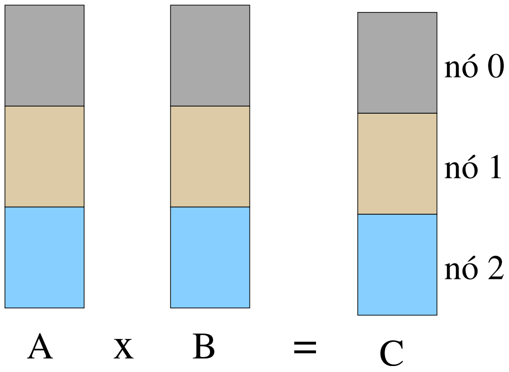

Figura 4.2: Computação do vetor resultante

tilhada oferecido por MOMEMTO é implementado dentro do kernel, o número de trocas de contexto e a latência no recebimento de interrupções é menor quando comparado a TreadMarks. Nós acreditamos que este ganho deve permanecer significativo mesmo quando MOMEMTO implementar um protocolo de consistência relaxada como o usado em TreadMarks.

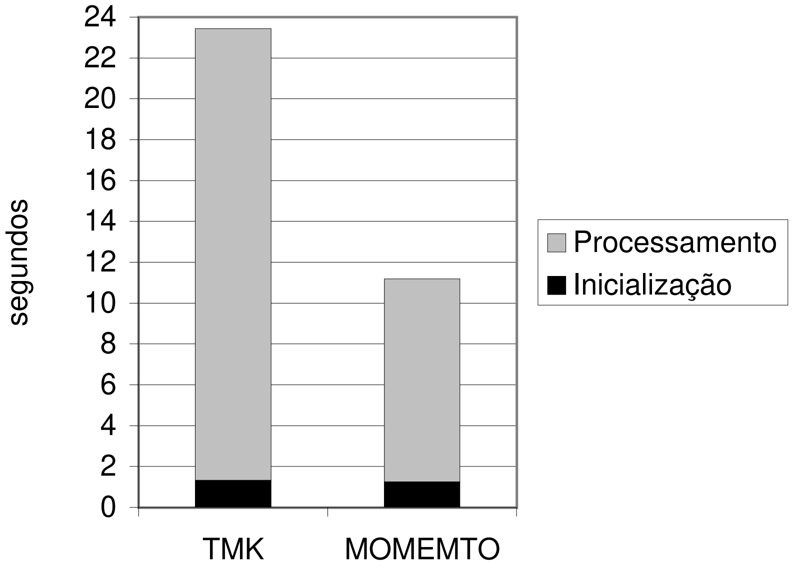

Figura 4.3: MOMEMTO x TMK

Falhas de páginas em MOMEMTO ocorrem sob demanda, ou seja, somente quando um nó acessar uma página do vetor que ele não seja a residência é que vai ocorrer a busca da página. Após o nó calcular todos os elementos desta página irá ocorrer a atualização da mesma. Deste modo, dentro do tempo de processamento, que foi de 9,96s, está o tempo gasto em falhas de páginas, atualizações e duas barreiras, as quais foram utilizadas para sincronizar a execução após os nós inicalizarem os seus vetores e após o cálculo do vetor resultante. Estes tempos são mostrados na Tabela 4.1. O tempo gasto no cálculo do vetor foi de 0,55s.

	Número	Total	Rede	Média
Atualizações	4000	2,78s	2,75s	694 μs
Falhas de Páginas	8000	6,63s	6,58s	828 μs
Barreiras	2	0,0010s	0,0010s	500 μs

Tabela 4.1: Tempo de processamento para MV Pequeno para cada nó

O campo *Média* representa o tempo médio para cada operação. No total foram realizadas 4.000 atualizações de páginas (uma terça parte do vetor C) pelo nó que realizou o cálculo. O nó 3, que é a residência do vetor C, não realizou atualização de página. O número total de falhas de páginas por nó foi de 8.000, ou seja, como cada nó realiza o cálculo de uma terça parte do vetor resultante, todos os nós tomaram falhas de páginas de uma terça parte dos dois vetores que eles não são a residência. A diferença entre o tempo $Total$ e o de $Rede$ foi o tempo gasto executando o código de MOMEMTO. Assim, o tempo para atualização foi de 1% e para falha de página de 1,47% do tempo $Total$. Estes tempos mostram que o *overhead* que MOMEMTO impõe é mínimo quando comparado à latência da rede, a qual pode ser melhorada utilizando outras tecnologias de comunicação como Myrinet ou Giganet.

O tempo gasto na barreira foi um pouco elevado devido ao fato de cada nó quando chega em cada barreira abrir uma conexão TCP com o gerente. Pode-se melhorar este tempo utilizando técnicas mais sofisticadas como discutido em outros trabalhos [17].

Este primeiro experimento reforça nosso argumento de que o desempenho da aplicação pode ser melhorado significantemente através de protocolos mais específicos para manter a coerência de memória.

Em nosso segundo teste com multiplicação de vetores, nós estudamos se MOMEMTO poderia realmente trabalhar com uma grande quantidade de memória. O experimento requer 825 MB de memória compartilhada. TreadMarks não conseguiu executar com esta carga em nosso ambiente de teste e, por esta razão, apresentamos os resultados somente para MOMEMTO na Tabela 4.2.

O tempo total de execução para MOMEMTO foi de 63,2s para esta carga. Como no teste anterior, o tempo total de processamento inclui o tempo para atualizações, falhas de

	Total
Inicialização	2,73s
Processamento	60,47s

Tabela 4.2: Tempo de execução para MV Grande

páginas e duas barreiras. Estes tempos são mostrados na Tabela 4.3. O tempo gasto realmente no cálculo do vetor foi de 5,35s e o restante (55,12s) foram custos da paralelização.

	Número	Total	Rede	Média
Atualizações	23466	16,34s	16,22s	696 μs
Falhas de Páginas	46932	38,78s	38,15s	826 μs
Barreiras	2	0,0012s	0,0011s	600 μs

Tabela 4.3: Tempo de processamento para MV Grande

Cada nó realizou 23.466 falhas de página (um terço do vetor C). Cada nó gerou 46.932 falhas de páginas, correspondente aos dois vetores para o qual o nó não era a residência. O tempo médio gasto em MOMEMTO foi $5,3$ μs para atualização e $13,3$ μs para falha de página.

Nestes experimentos preliminares, o protocolo simples de consistência utilizado por MOMEMTO permite à aplicação decidir sobre a política e o protoloco utilizado na atualização. Esta característica pode ter contribuído para o melhor desempenho do sistema quando comparado com TreadMarks.

Com o intuito de avaliar o mecanismo em kernel oferecido por MOMEMTO, comparamos a mesma aplicação com uma implementação utilizando o paradigma de troca de mensagens. As Figuras 4.4 e 4.5 mostram o tempo de execução para a multiplicação de vetores com MOMEMTO e a mesma aplicação implementada em MPI[5].

Como no teste anterior, em ambos sistemas cada nó calcula uma terça parte do vetor C, cada nó é a residência de um vetor e o inicializa na execução. Na versão em MPI, mensagens são utilizadas para distribuir as partes utilizadas por outros nós do vetor A e B, ou seja, o nó 0 envia uma terça parte do seu vetor para o nó 1 e uma outra terça parte para o nó 2. O mesmo processo ocorre no nó 1 sobre o seu vetor, B. No final, o nó que é a residência do vetor C recebe mensagens com os resultados dos outros nós.

Na Figura 4.4, cada vetor tem 12.000 páginas (MV Pequeno). Embora MOMEMTO mostre um aumento de 3,5% no tempo de execução quando comparado com MPI, o tempo de processamento diminuiu em 7,5% com MOMEMTO. A razão para o tempo gasto na

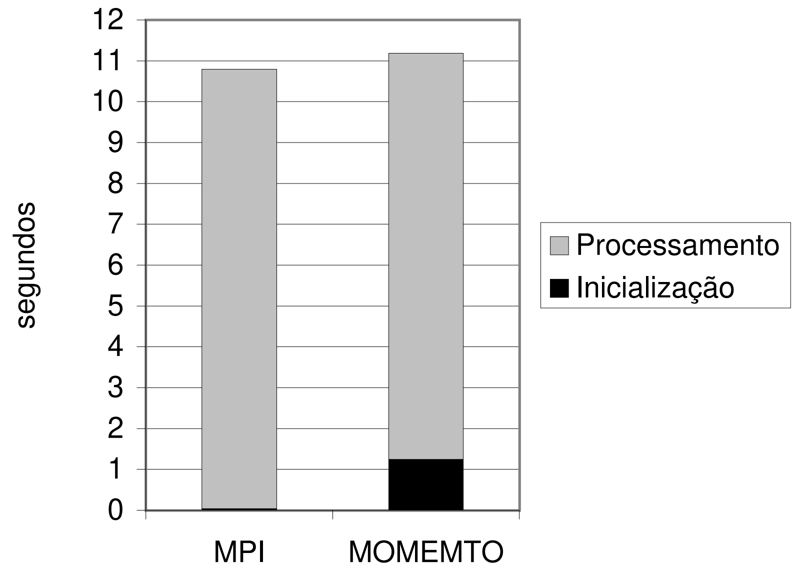

Figura 4.4: MOMEMTO x MPI para MV Pequeno

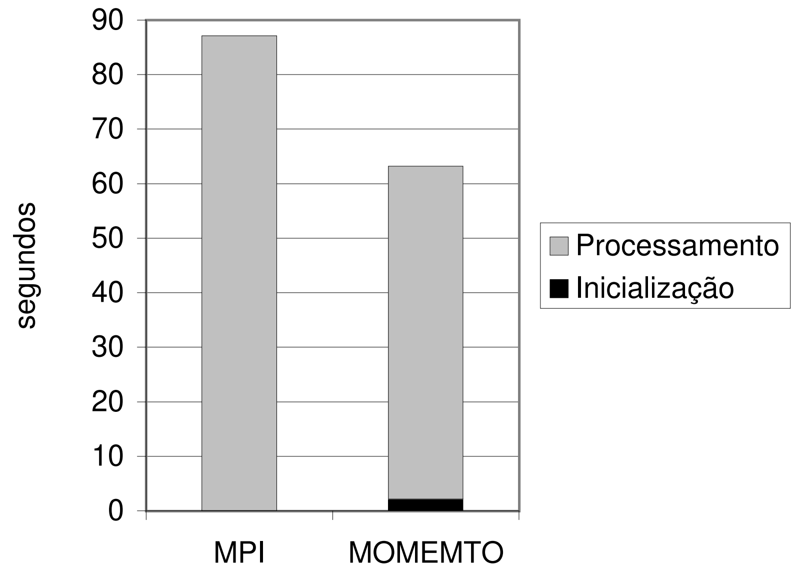

Figura 4.5: MOMEMTO x MPI para MV Grande

inicialização ser maior com MOMEMTO é devido ao fato dele forçar uma falha de página na inicialização para cada uma que o nó é a residência.

No teste MV Grande, apresentado na Figura 4.5, onde cada vetor tem 70.398 páginas, o tempo de execução diminuiu 27% na versão com MOMEMTO.

Em geral, analisando o número de mensagens, MPI realizou menos trocas que MOMEMTO. No pior caso, MOMEMTO precisa de 3 vezes mais mensagens que MPI. Isto ocorre para uma página que o nó não é a residência e ele precisa acessar uma página do vetor C, sofrendo uma falha de página e atualizando a mesma após o cálculo. Falhas de página no MOMEMTO gastam 2 mensagens em modo kernel e a atualização é feita através de uma chamada de sistema e uma mensagem enviada também em modo kernel. A Tabela 4.4 mostra os tempos de comunicação com MOMEMTO e MPI. Este tempo no MOMEMTO é gasto realizando atualizações e falhas de páginas, enquanto que em MPI é recebendo e enviando mensagens dos vetores A e B e, enviando e recebendo mensagens com o resultado do vetor C.

	MV Pequeno	MV Grande
MPI	9,99s	82,53s
MOMEMTO	9,39s	55,12s

Tabela 4.4: Tempo de comunicação

No teste MV Pequeno, MOMEMTO foi 6% mais rápido que MPI e no teste MV Grande reduziu o tempo de comunicação em 33% comparando com MPI.

Novamente, nós argumentamos que o melhor desempenho de MOMEMTO é relacionado com o baixo *overhead* de suas operações em kernel. MOMEMTO sofre menos trocas de contexto e tem a latência nas interrupções menor que a versão em MPI.

4.4 IS - Integer Sort

A aplicação Integer Sort (IS) [18] ordena um vetor de N inteiros usando chaves no intervalo [0,Bmax] através da técnica *Bucket Sort*. As chaves são divididas igualmente entre os processadores do sistema. As principais estruturas de dados são um vetor de chaves e um vetor que indica a densidade das chaves a serem ordenadas. A computação é dividida em N fases, onde N é o número de processadores executando a aplicação.

Na versão MPI foi utilizada a implementação do IS que vem com o *NAS Parallel Benchmarks 2.3* [18]. Com TreadMarks foi utilizada a implementação distribuída do IS baseada em barreiras da versão 1.0.3.2. A implementação em MOMEMTO foi baseada no algoritmo utilizado na versão TreadMarks, pois os dois utilizam memória compartilhada. Foi necessário, entretanto, utilizar a chamada de sistema $mmto\_sync$ para, quando necessário, realizar a coerência dos dados compartilhados.

Foram realizados testes com duas entradas para IS, CLASS A e CLASS B, como mostrado na Tabela 4.5.

CLASS A	CLASS B
$N = 2^{23}$	$N = 2^{25}$
$Bmax = 2^{19}$	$Bmax = 2^{21}$
10 iterações	10 iterações

Tabela 4.5: Entrada utilizada para o IS

4.4.1 CLASS A

A Figura 4.6 mostra o tempo total de execução para as três implementações. A execução do IS CLASS A, utilizando o algoritmo sequencial que acompanha o NAS foi de 14,11s. Utilizando MOMEMTO houve uma redução de 31% e 18% no tempo de execução em relação ao Tmk e MPI, respectivamente, em 2 nós. Por sua vez, com 4 nós a redução foi de 54% quando comparado a Tmk e 19% a MPI. Finalmente, para 8 nós MOMEMTO reduziu em 79% o tempo de execução em relação a Tmk e 1,27% em comparação a MPI.

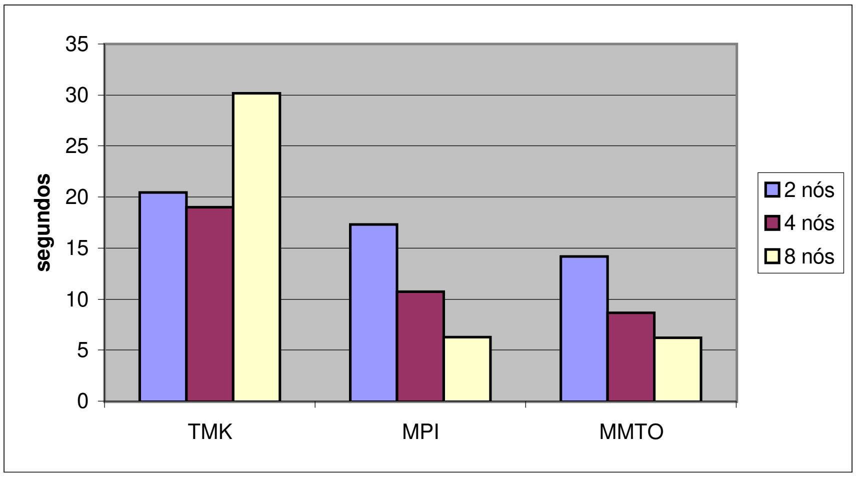

Figura 4.6: MOMEMTO x TMK x MPI - Class A

4.4.1.1 MOMEMTO

Discutiremos primeiro o resultado para MOMEMTO. A Tabela 4.6 mostra a quantidade de bytes e o número de mensagens gasto durante a execução da aplicação.

O *Número de mensagens* e *Bytes trafegados* englobam as mensagens utilizadas pelo sistema (*overhead*) e os dados úteis da aplicação. O tempo de comunicação foi o tempo

	2 nós	4 nós	8 nós
Número de mensagens	3.584	6.912	11.648
Bytes trafegados	12MB	21 MB	31.5MB
Tempo de comunicação	1,42s	2,81s	4,80s

Tabela 4.6: Estatísticas do MOMEMTO - CLASS A

gasto realizando barreiras, atualizações e falhas de páginas. A Tabela 4.7 detalha o tempo de comunicação.

	2 nós	4 nós	8 nós
Número de atualizações	2560	2560	2560
Tempo em atualizações	0,71s	1,04s	1,23s
Número de falhas de páginas	512	1536	3584
Tempo gasto em falhas de páginas	0,51s	1,48s	2,97s
Tempo gasto em barreiras	0,20s	0,29s	0,60s

Tabela 4.7: Comunicação no MOMEMTO - CLASS A

O número de atualizações se mantém quando se aumenta o número de nós. Isto mostra um desbalanceamento do algoritmo da aplicação, visto que sempre o nó zero realiza mais processamento que os demais. Entretanto, o seu tempo aumenta, pois o nó zero tem que enviar as atualizações para mais nós.

O número de falhas de página aumenta de acordo com o número de nós, pois o número de páginas que cada nó será a residência diminui, gerando assim mais falhas de páginas para acessar a área compartilhada. O tempo gasto em barreira também sofre um aumento, o que era esperado, visto que o algoritmo utilizado não é escalável, pois cada nó abre uma conexão com o gerente quando chega à uma barreira.

Como MOMEMTO não impõe um modelo de consistência de memória, para cada atualização realizada pela aplicação apenas 4 bytes são adicionados pelo sistema, que são utilizados para informar qual página receberá os dados atualizados. O resto dos bytes trafegados são dados úteis da aplicação e é gerada apenas uma mensagem para esta operação. Por sua vez, na falha de página são utilizadas duas mensagens: uma com 4 bytes para especificar a página requisitada e outra para receber os dados referentes à ela.

4.4.1.2 TreadMarks

A Tabela 4.8 demonstra as estatísticas para TreadMarks.

Como podemos observar, o número de mensagens e a quantidade de bytes trafegados

	2 nós	4 nós	8 nós
Número de mensagens	21.093	84.451	223.155
Bytes trafegados	41.1MB	192.9MB	733MB
Falhas de Páginas	5.632	16.512	45.760
Diffs	4.864	15.232	34.240
Twins	514	617	585

Tabela 4.8: Estatísticas para TreadMarks - CLASS A

cresce vertiginosamente quando aumentamos o número de nós. Este comportamento é devido aos efeitos do protocolo de coerência de Tmk que, quando aumenta o número de nós, aumenta a quantidade de falhas de página e *diffs* gastos pelo sistema.

4.4.1.3 MPI

Como demonstra a Tabela 4.9, quando aumentamos o número de nós a aplicação transfere a mesma soma de bytes, mas utiliza mais mensagens. O tempo de processamento e comunicação também é diminuído, mostrando um balanceamento da aplicação.

	2 nós	4 nós	8 nós
Tempo de processamento	7,22s	3,63s	1,99s
Tempo de comunicação	10,10s	7,11s	4,27s
Bytes trafegados	171MB	171MB	171MB
Número de mensagens	110	484	2.024

Tabela 4.9: Estatísticas para o MPI - CLASS A

4.4.1.4 Discussão

Como podemos observar, MOMEMTO reduziu em média 89,86% o número de mensagens em relação a Tmk. Está redução ocorreu basicamente devido a MOMEMTO não impor um modelo de consistência e deixar o mesmo a cargo do programador. Em Tmk, para 2 nós, foram gastas 4.864 mensagens em *diffs* e 5.632 em falhas de página, enquanto MOMEMTO não realizou nenhum *diff* e foram geradas apenas 512 falhas de páginas, visto que, após o primeiro acesso o nó manteve a página em memória e não ocorreram mais falhas a ela. Outro ponto a salientar é que o padrão da aplicação é único escritor (*single-writer*) enquanto Tmk é múltiplos escritores (*multiple-writer*). Tmk tem um alto *overhead* para permitir múltiplos escritores para esta aplicação.

Quando comparamos nosso sistema com MPI, notamos que MPI transfere mais dados e utiliza menos mensagens. Como o algoritmo das duas implementações são bem distintos fica difícil uma comparação mais profunda sobre ambos. Entretanto, podemos observar que o mesmo algoritmo de Tmk quando utilizado com MOMEMTO deixa os tempos competitivos com MPI e sinaliza que MOMEMTO pode obter resultados competitivos a MPI.

4.4.2 CLASS B

A Figura 4.7 mostra os tempos de execucão para MOMEMTO, Tmk e MPI rodando a aplicação IS para 2, 4 e 8 nós. A execução do IS CLASS B utilizando o algoritmo sequencial distribuído junto com o NAS foi de 66,84s.

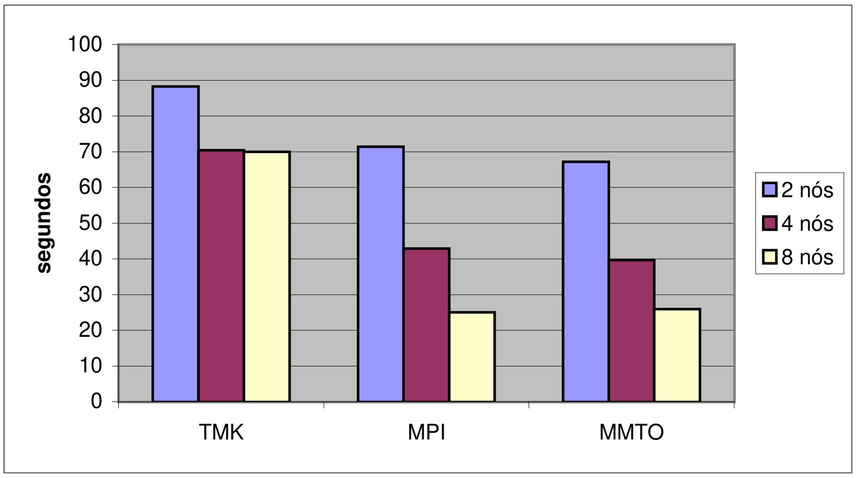

Figura 4.7: MOMEMTO x TMK x MPI - Class B

4.4.2.1 MOMEMTO

A Tabela 4.10 apresenta o número de mensagens geradas por MOMEMTO e a quantidade de bytes trafegados. Aumentando o número de nós, cresce o número de mensagens geradas pelo sistema e a quantidade de bytes trafegados. Este comportamento está relacionado ao aumento do número de falhas de páginas no sistema. Cada nó foi a residência de menos páginas quando aumentamos o número de nós, gerando assim mais mensagens e bytes trafegados pelo sistema. Na Tabela 4.11, são detalhados estes valores.

O aumento no tempo de barreira é devido ao fato do sistema a cada barreira abrir uma conexão com o gerente, e quando aumentamos o número de nós geramos mais mensa-

	2 nós	4 nós	8 nós
Número de mensagens	14.336	17.408	31.232
Bytes trafegados	48MB	88MB	132MB

Tabela 4.10: Estatísticas do MOMEMTO - CLASS B

	2 nós	4 nós	8 nós
Tempo em atualizações	3,08s	4,68s	5,48s
Número de falhas de páginas	2.048	6.144	14.336
Tempo gasto em falhas de páginas	2,08s	5,74s	11,93s
Tempo gasto em barreiras	0,04s	0,09s	0,74s

Tabela 4.11: Comunicação no MOMEMTO - CLASS B

gens. Sabemos que este algoritmo não é escalável e pode ser melhorado utilizando outros métodos descritos na literatura.

4.4.2.2 TreadMarks

A Tabela 4.12 mostra as estatísticas para TreadMarks.

	2 nós	4 nós	8 nós
Número de mensagens	84.069	292.383	705.373
Bytes trafegados	164MB	571MB	1375MB
Falhas de Páginas	22.528	105.472	269.824
Diffs	19.456	40.448	81.664
Twins	2.053	2.359	2.168

Tabela 4.12: Estatísticas para TreadMarks - CLASS B

Como podemos observar, Tmk não tem um bom desempenho quando aumentamos o número de nós. O principal problema está no número de *diffs*, falhas de páginas e a quantidade de bytes trafegados. Do total de bytes transmitidos em 8 nós (1375 MB), 4 MB são dados de Tmk, ou seja, informações para manter a coerência dos dados da aplicação e dados internos.

4.4.2.3 MPI

Na Tabela 4.13 são demonstrados os tempos relativos à execução de MPI.

O número de mensagens aumentou, como esperado, quando aumentou o número de nós. Entretanto, a quantidade de bytes trafegados permaneceu a mesma.

	2 nós	4 nós	8 nós
Tempo de processamento	28,87s	14,54s	7,89s
Tempo de comunicação	42,5s	28,36s	17,11s
Bytes trafegados	469MB	469MB	469MB
Número de mensagens	110	484	2.024

Tabela 4.13: Estatísticas para o MPI - CLASS B

4.4.2.4 Discussão

MOMEMTO reduziu o tempo de execução quando comparando a Tmk em 23%, 43% e 63% para 2, 4 e 8 nós respectivamente. Novamente podemos observar que a quantidade de bytes e o número de mensagens em MOMEMTO é bastante inferior a Tmk. MOMEMTO não utiliza *diff* ou *twin*, e deixa a consistência a cargo do programador. Deste modo, oferece mais flexibilidade ao programador e abre espaço para otimizações. Com esta avaliação preliminar, podemos concluir que MOMEMTO consegue agregar a memória dos nós no cluster de forma eficiente, permitindo à aplicação rodar com uma grande quantidade de memória compartilhada e melhorar sua execução através de primitivas básicas de sincronização com baixo *overhead*.

Capítulo 5

Trabalhos Relacionados

Diversos estudos têm examinado métodos para usar a memória remota em um cluster. Aplicações comuns são o uso da rede como memória para *swap* e sistemas de memória compartilhada distribuída que provêem uma abstração de memória compartilhada sobre a rede. Trabalhos relacionados a estas duas técnicas serão descritos a seguir.

5.1 *Swap* Sobre a Rede

O GMS [21] (*Global Memory Service*) realiza um gerenciamento de memória distribuído no cluster. Este sistema adiciona em cada nó uma área de memória global, onde esta cresce ou diminui de acordo com a carga do nó. A idéia básica é prover *swap* sobre a rede para as aplicações e assim eliminar a maioria dos acessos ao disco. Ele utiliza nós ociosos para receber páginas de nós muito ativos. Entretanto, ele não utiliza informações sobre a aplicação para fazer a política de substituição e não tenta otimizar a localização de páginas compartilhadas. Ele difere da nossa proposta pois não existe o compartilhamento de dados entre os nós que executam a aplicação.

5.2 Memória Compartilhada Distribuída

Cashmere-VLM [19] é um software DSM que realiza *swap* sobre a rede. Ele procura aumentar a área compartilhada da aplicação liberando memória do nó quando ele está sobrecarregado, ou seja, libera páginas de um nó levando em consideração as cópias que existem nos outros nós do cluster. Cashmere-VLM é implementado no nível do usuário e utiliza um modelo de consistência fixo. Uma característica interessante de Cashmere-VLM é que ele permite a uma página trocar de *home* dinamicamente.

O trabalho relacionado mais próximo ao nosso é JIAJIA [23], que é um software DSM onde a memória física dos computadores são combinadas em um grande espaço compartilhado. Este sistema utiliza um protocolo de coerência baseado em *locks*. Novamente, MOMEMTO difere deste trabalho no fato que nosso sistema compartilha páginas dentro do kernel e não força um protocolo de coerência, permitindo mais flexibilidade para a aplicação.

Capítulo 6

Conclusões

Nesta tese foi apresentado o sistema MOMEMTO (*MOre MEMory Than Others*), um novo conjunto de mecanismos de kernel que suportam memória compartilhada global nos clusters. MOMEMTO foi concebido para oferecer mais flexibilidade em um sistema de memória compartilhada e para explorar todas as memórias distribuídas dos nós do cluster.

Oferecendo primitivas básicas de sincronização, MOMEMTO permite a aplicação ter um maior controle sobre o uso de sua memória e abre espaço para otimizações feitas especificamente para cada aplicação.

MOMEMTO mostrou um baixo *overhead* de execução em nossos experimentos e pode realmente executar de maneira satisfatória para aplicações que requeiram uma grande quantidade de memória compartilhada, introduzindo baixo *overhead* no sistema de gerenciamento de memória virtual no kernel. É necessário, entretanto, estender significativamente o escopo dos experimentos para avaliar melhor o impacto de MOMEMTO sobre diferentes tipos de aplicações.

Em trabalhos futuros espera-se analisar o impacto da utilização de vários modelos de consistência de memória para determinadas classes de aplicações, além de realizar testes do sistema com diferentes classes de aplicações.

Referências Bibliográficas

[1] "NERSC. M-VIA". http://www.nersc.gov/research/FTG/via.

[2] "Rocketcalc". http://www.rocketcalc.com.

[3] "NCSA Urbana-Champaign". http://www.ncsa.uiuc.edu, 2001.

[4] "CBRC - Tsukuba Advanced Computing Center - TACC/AIST Tsukuba". http://www.cbrc.jp/magi/, 2001.

[5] "Message Passing Interface Forum". http://www.mpi-forum.org.

[6] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly & Associates, 2nd ed., June 2001.

[7] Amza, C., Cox, A. L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W. "TreadMarks: Shared Memory Computing on Networks of Workstations". *IEEE Computer*, v. 29, n. 2, pp. 18–28, 1996.

[8] Amza, C., Cox, A. L., Dwarkadas, S., Zwaenepoel, W. "Software DSM Protocols that Adapt between Single Writer and Multiple Writer". In: *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*, pp. 261–271, 1997.

[9] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, D., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., Weeratunga, S. K. "The NAS Parallel Benchmarks". *The International Journal of Supercomputer Applications*, v. 5, n. 3, pp. 63–73, Fall 1991.

[10] Bal, Heri E., Kaashoek, M. Frans, Tanenbaum, Andrew S. "Orca: a language for parallel programming of distributed systems". *IEEE Transactions on Software Engineering*, v. 18, n. 3, pp. 190–205, 1992.

[11] Bershad, Brian N., Zekauskas, Matthew J. *Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors*. Technical Report CMU-CS-91-170, Pittsburgh, PA (USA), 1991.

[12] Bianchini, R., Kontothanassis, L. I., Pinto, R., De Maria, M., Abud, M., Amorim, C. L. "Hiding Communication Latency and Coherence Overhead in Software DSMs". In: *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pp. 198–209, 1996.

[13] Bianchini, R., LeBlanc, T., Veenstra, J. "Categorizing Network Traffic in Update-Based Protocols on Scalable Multiprocessors". In: *In Proceedings of the International Parallel Processing Symposium*, April 1996.

[14] Brin, Sergey, Page, Lawrence. "The anatomy of a large-scale hypertextual Web search engine". *Computer Networks and ISDN Systems*, v. 30, n. 1–7, pp. 107–117, 1998.

[15] Cox, A., de Lara, E., Hu, W. Zwaenepoel Y. C. "A Performance Comparison of Homeless and Home-Based Lazy Release Consistency Protocols for Software Shared Memory". In: *Proc. of the 5th IEEE Symp. on High-Performance Computer Architecture (HPCA-5)*, 1999.

[16] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, 2nd ed., December 2002.

[17] David E. Culler, Anoop Gupta, Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, August 1998.

[18] David H. Bailey and Leonardo Dagum and E. Barszcz and Horst D. Simon. "NAS Parallel Benchmark Results". In: *Supercomputing*, pp. 386–393, 1992.

[19] Dwarkadas, S., Hardavellas, N., Kontothanassis, L., Nikhil, R., Stets, R. "Cashmere-VLM: Remote Memory Paging for Software Distributed Shared Memory". In: *Proc., IPPS*, April 1999.

[20] Engler, Dawson R., Kaashoek, M. Frans, O'Toole, James. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Symposium on Operating Systems Principles*, pp. 251–266, 1995.

[21] Feely, Michael J., Morgan, William E., Pighin, Frederic H., Karlin, Anna R., Levy, Henry M., Tekkath, Chandramohan A. “Implementing Global Memory Management in a Workstation Cluster”. In: *Proceedings of the 15th Symposium on Operating Systems Principles*, pp. 201–212, December 1995.

[22] Geist, G. A., Sunderam, V. S. “Network-based Concurrent Computing on the PVM System”. *Concurrency: practice and experience*, v. 4, n. 4, pp. 293–312 (or 293–311??), 1992.

[23] Hu, Weiwu, Shi, Weisong, Tang, Zhimin. “JIAJIA: An SVM System Based on A New Cache Coherence Protocol”. In: *In Proceedings of the High Performance Computing and Networking (HPCN'99)*, pp. 463–472, April 1999.

[24] Iftode, L., Singh, J. P. “Shared Virtual Memory: Progress and Challenges”. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, v. 87, n. 3, pp. 498–507, 1999.

[25] Iftode, L., Singh, J. P., Li, K. “Scope Consistency: A Bridge between Release Consistency and Entry Consistency”. In: *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pp. 277–287, 1996.

[26] Karlsson, Magnus, Stenström, Per. “Effectiveness of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared-Memory Systems”. *Journal of Parallel and Distributed Computing*, v. 43, n. 2, pp. 79–93, 1997.

[27] Keleher, Pete, Cox, Alan L., Zwaenepoel, Willy. “Lazy Release Consistency for Software Distributed Shared Memory”. In: *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pp. 13–21, 1992.

[28] Li, K., Hudak, P. “Memory Coherence in Shared Virtual Memory Systems”. In: *PODC86*, pp. 229 – 239, August 1986.

[29] Li, K., Hudak, P. “Memory Coherence in Shared Virtual Memory Systems”. In: *ACM Transactions on Computer Systems*, pp. 321–359, November 1989.

[30] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley Professional, 1996.

[31] Monnerat, L. R., Bianchini, R. “Efficiently Adapting to Sharing Patterns in Software DSMs”. In: *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, 1998.

[32] Monnerat., L.R.R. “Adaptação Eficiente a Padrões de Compartilhamento em Software DSMs.”. *Tese de Mestrado, COPPE/UFRJ*, Dezembro 1997.

[33] Mowry, T. C., Chan, C., Lo, A. “Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory”. In: *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pp. 300–311, 1998.

[34] Pinto, R. *Estratégias de Software e Hardware para Otimização de Sistemas de Memória Compartilhada Distribuída*. PhD thesis, Federal University of Rio de Janeiro, Brazil, December 2001.

[35] Sun Microsystems Inc. “SGE”. http://wwws.sun.com/software/gridware.

[36] Trevisan, Thobias S., Costa, Vitor S., Whately, L., , Amorim, C. L. “Distributed Shared Memory in Kernel Mode”. In: *Proceedings of the 14th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*, pp. 159 – 166, October 2002.

[37] Uresh Vahalia . *UNIX Internals: The New Frontiers*. Prentice Hall, October 1995.

[38] Veridian Systems. “OpenPBS”. http://www.openpbs.org.

[39] Whately, L., Pinto, R., Rangarajan, M., Iftode, L., Bianchini, R., Amorim, C. L. “Adaptive Techniques for Home-Based Software DSMs”. In: *Proceedings of the 13th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*, pp. 164 – 171, September 2001.

[40] Zhou, Y., Iftode, L., Li, K. “Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems”. In: *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pp. 75–88, 1996.

Apêndice A

Fonte do programa multiplicação de vetores

```
/* Multiplicação de vetores
 *
 * Última atualização Nov 18 11:53:26 BRT 2002
 *
 * Thobias Salazar Trevisan <thobias@cos.ufrj.br>
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <math.h>

#include "MMTO.h"

//#define PAGES 27   // Pages Total
#define PAGES   36000   // Pages total
//#define PAGES 211194   // Pages Total
//#define PAGES 315000   // Pages Total
#define ELEM PAGES*1024 // Num elementos total
#define DIF(x,y) (((double)y.tv_sec - x.tv_sec) * 1000000 + ((double)y.tv_usec -
x.tv_usec)) / 1000000

main(int argc, char *argv[])
{
int *A, *B, *C, i, j;
int n=0;
double lat0, lat1;
struct timeval start, inicializa, stop;

gettimeofday(&start, 0);
A = (int *)mmto_mmap(0,PAGES*4096,0);
B = &A[ELEM/3];
C = &A[(ELEM/3)*2];

mmto_load("hosts.txt");
printf("Terminou o mmto_load\n");
n=MMTO_start_barrier(4);
```

```
        if (n<0) printf("ERRO MMTO_start_barrier\n");

n=MMTO_cheguei_barrier(1);
        if (n<0) printf("ERRO MMTO_cheguei_barrier\n");
gettimeofday(&inicializa, 0);
////////////////////////////////////////////////////////////
// inicializacao dos vetores A B C
///////////////////////////////////////////////////////////

if (host->whoami == 0)
       for (i=0;i<ELEM/3;i++) A[i]=i;

if (host->whoami == 1)
for (i=0;i<ELEM/3;i++) B[i]=2;


if (host->whoami == 2)
for (i=0;i<ELEM/3;i++) C[i]=0;
printf("Termino da inicializacao do vetor\n");
///////////////////////////////////////////////////////////////
// Calculo do vetor C
/////////////////////////////////////////////////////////////

n=MMTO_cheguei_barrier(2);
if (n<0) printf("ERRO MMTO_cheguei_barrier\n");

if (host->whoami == 0){
for (i=0;i<ELEM/9;i++){
C[i]=B[i]*A[i];
//if ((i%1024) == 0) printf("%d ",C[i]);
if ((i%1024==0) && (i!=0)){
n=mmto_sync((unsigned long)C+(((i/1024-1))*4096),4096,MMTO_UPDATE);
if (n<0)
                  printf("ERRO update\n");
n=mmto_sync((unsigned long)C+(((i/1024-1))*4096),4096,MMTO_FREE);
if (n<0)
                  printf("ERRO free C\n");
n=mmto_sync((unsigned long)B+(((i/1024-1))*4096),4096,MMTO_FREE);
if (n<0)
                  printf("ERRO free B\n");
}
}
printf("Comecei a enviar os updates\n");
n=mmto_sync((unsigned long)C+((PAGES/9-1)*4096),4096,0);
if (n<0)
                printf("ERRO update segundo\n");
}

if (host->whoami == 1){
for (i=ELEM/9;i<((ELEM/9)*2);i++){
C[i]=B[i]*A[i];
//if ((i%1024) == 0) printf("%d ",C[i]);
if (i%1024==0){
n=mmto_sync((unsigned long)C+(((i/1024-1))*4096),4096,MMTO_UPDATE);
if (n<0)
                  printf("ERRO update\n");
```

```
n=mmto_sync((unsigned long)C+(((i/1024-1))*4096),4096,MMTO_FREE);
if (n<0)
                  printf("ERRO free C\n");
n=mmto_sync((unsigned long)A+(((i/1024-1))*4096),4096,MMTO_FREE);
if (n<0)
                  printf("ERRO free A\n");
}
}
printf("Comecei a enviar os updates\n");
n=mmto_sync((unsigned long)C+((((PAGES/9)*2)-1)*4096),4096,0);
if (n<0)
                printf("ERRO update segundo\n");
}

if (host->whoami == 2){
for (i=((ELEM/9)*2);i<ELEM/3;i++){
C[i]=B[i]*A[i];
//if ((i%1024) == 0) printf("%d ",C[i]);
if (i%1024==0){
n=mmto_sync((unsigned long)A+(((i/1024-1))*4096),4096,MMTO_FREE);
if (n<0)
                  printf("ERRO free A\n");
n=mmto_sync((unsigned long)B+(((i/1024-1))*4096),4096,MMTO_FREE);
if (n<0)
                  printf("ERRO free B\n");
}
}
}

////////////////////////////////////////////////////////////
// Final do Calculo do vetor C
////////////////////////////////////////////////////////////
printf("Termino do calculo do vetor\n");
n=MMTO_cheguei_barrier(3);
        if (n<0) printf("ERRO MMTO_cheguei_barrier\n");
////////////////////////////////////////////////////////////
// Imprime o vetor C
////////////////////////////////////////////////////////////

if (host->whoami == 2)
for (i=0;i<ELEM/3;i++){
n=C[i];
//if ((i%1024) == 0)
//printf("%d ",C[i]);
}

n=MMTO_cheguei_barrier(4);
if (n<0) printf("ERRO MMTO_cheguei_barrier\n");

gettimeofday(&stop, 0);
lat0= DIF(start, inicializa);
lat1= DIF(start, stop);
printf("\nLatencia inicializacao : %f   total %f\n",lat0, lat1);
}
```