COPPE/UFRJ

MEMÓRIA GLOBAL PARA CLUSTERS DE COMPUTADORES ATRAVÉS DE MECANISMO DE KERNEL

Daniel da Cunha Schmidt

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Claudio Luis de Amorim

Rio de Janeiro
Setembro de 2010

MEMÓRIA GLOBAL PARA CLUSTERS DE COMPUTADORES ATRAVÉS DE MECANISMO DE KERNEL

Daniel da Cunha Schmidt

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) 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.

Examinada por:

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Cristiana Bentes, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

SETEMBRO DE 2010

Schmidt, Daniel da Cunha

Memória Global para Clusters de Computadores através de Mecanismo de Kernel/Daniel da Cunha Schmidt. – Rio de Janeiro: UFRJ/COPPE, 2010.

XI, 47 p.: il.; 29, 7cm.

Orientador: Claudio Luis de Amorim

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2010.

Referências Bibliográficas: p. 45 – 47.

1. Software DSM. 2. Sistemas Operacionais. 3. *Cluster* de Computadores. I. Amorim, Claudio Luis de. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Aos meus pais e aos meus irmãos.

Agradecimentos

Gostaria de agradecer ao meu orientador Claudio Amorim, por me mostrar os caminhos a serem seguidos e ajudar a esclarecer as muitas dúvidas que surgiram ao longo desses anos.

Ao Lauro e ao Diego, pela atenção e boa vontade em me ajudar nos mais diversos problemas.

Ao Almir, Arthur, Bruno, Lawrence, Flávio, Jesus, e aos Alexandres, por estarem sempre presente, seja na biblioteca, estudando, ou numa mesa de bar, ouvindo minhas reclamações ou simplesmente jogando conversa fora.

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

MEMÓRIA GLOBAL PARA CLUSTERS DE COMPUTADORES ATRAVÉS DE MECANISMO DE KERNEL

Daniel da Cunha Schmidt

Setembro/2010

Orientador: Claudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

Nesta dissertação é proposto um sistema que implementa um mecanismo no kernel do Linux que oferece às aplicações toda a memória distribuída disponível num cluster de computadores, permitindo a utilização de mais memória do que a localmente disponível em cada computador. A implementação de um protótipo foi feita no kernel 2.6 do Linux, onde o sistema de gerenciamento de memória virtual foi modificado para que falhas de página em um endereço de memória compartilhado buscassem os dados nos outros nós do cluster. Espera-se que aplicações *out of core* sejam diretamente beneficiadas por esse sistema, dado que elas podem trocar a latência de acesso ao disco pela latência da rede. Para não se limitar apenas ao compartilhamento global de memória no cluster, foram implementadas primitivas de sincronização que permitem ao programador realizar a manutenção da coerência dos dados em aplicações paralelas. Para avaliar a implementação do mecanismo proposto, testes foram realizados e seus resultados discutidos.

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

GLOBAL MEMORY FOR COMPUTER CLUSTER BY KERNEL ENGINE

Daniel da Cunha Schmidt

September/2010

Advisor: Claudio Luis de Amorim

Department: Systems Engineering and Computer Science

This dissertation proposes a system that implements a mechanism in the Linux kernel that offers to the application all the available distributed memory in a cluster of computers, allowing the use of more memory than that is available locally at each computer. The implementation of a system was made in the Linux 2.6 kernel, where the system virtual memory management was modified so that page faults in a shared memory address will retrieve data on other cluster nodes. It is expextede that *out of core* applications to be directly benefited by the system, as they can change disk access latency by network latency. To not be limited only to sharing global memory in the cluster, synchronization primitives were implemented to allow programmers to perform maintenance of data consistency in parallel applications. To evaluate the implementation of the proposed mechanism, tests were performed and their results discussed.

Sumário

Lista de Figuras x

Lista de Tabelas xi

1 Introdução 1

1.1 Motivação 1

1.2 Contribuições 4

1.3 Organização da Dissertação 4

2 Conhecimentos Básicos 5

2.1 Memória Compartilhada Distribuída em Software 5

2.1.1 Gerenciamento de sistemas SDSM 6

2.1.2 Algoritmos SDSM 7

2.1.3 Modelos de Consistência de Memória 9

2.2 O Sistema de Gerenciamento de Memória Virtual no Linux 11

2.2.1 Paginação e Memória Virtual 11

2.2.2 Espaço de Endereçamento de um Processo e Regiões de Memória 11

3 Implementação de Memória Global no Kernel 13

3.1 Projeto 13

3.2 Implementação 16

3.2.1 Dificuldades encontradas durante a implementação 16

3.2.2 Modificações no kernel 16

3.2.3 Preparação do sistema 18

3.2.4 Inicialização do sistema 18

3.2.5 Funcionamento do sistema 23

3.2.6 Encerramento do sistema 26

4 Análise Experimental 27

4.1 Ambiente de testes 27

4.2 Métricas de desempenho 27

4.3 Problemas encontrados durante os experimentos 28

4.4 Aplicações 28

4.4.1 Multiplicação de vetores 29

4.4.2 Mergesort 34

4.5 Análise 39

5 Trabalhos Relacionados 41

5.1 Swap Sobre a Rede 41

5.2 Memória Compartilhada Distribuída em Software 42

6 Conclusões e Trabalhos Futuros 43

6.1 Conclusões 43

6.2 Trabalhos Futuros 44

Referências Bibliográficas 45

Lista de Figuras

2.1 Esquema de funcionamento de um SDSM. 6
2.2 Regiões de memória e o espaço de endereçamento de um processo. . . 12

3.1 Espaço de endereçamento compartilhado. 13
3.2 Integração do sistema com o S.O. 14
3.3 Organização da memória do sistema. 21
3.4 Esquema de conexão dos sockets. 22

4.1 Tempo de execução sequencial para a multiplicação de vetores. 29
4.2 Tempo gasto em falhas de página para a multiplicação de vetores sequencial. 30
4.3 Tempo de execução paralela para a multiplicação de vetores. 32
4.4 Tempo gasto em falhas de página para a multiplicação de vetores paralela. 33
4.5 Tempo de execução sequencial para mergesort. 35
4.6 Tempo gasto em falhas de página para mergesort sequencial. 36
4.7 Tempo de execução paralela para mergesort. 37
4.8 Tempo gasto em falhas de página para mergesort paralelo. 38

Lista de Tabelas

4.1 Configuração das Máquinas 27
4.2 Tempos médios para a multiplicação de vetores sequencial 31
4.3 Número de Falhas e Mensagens para a multiplicação de vetores sequencial 31
4.4 Tempos médios para a multiplicação de vetores paralela 33
4.5 Número de Falhas e Mensagens para a multiplicação de vetores paralela 34
4.6 Tempos médios para mergesort sequencial 35
4.7 Número de Falhas e Mensagens para mergesort sequencial 36
4.8 Tempos médios para mergesort paralelo 38
4.9 Número de Falhas e Mensagens para mergesort paralelo 39
4.10 Tempo de busca. 39

Capítulo 1

Introdução

1.1 Motivação

Atualmente, apesar da computação em *clusters* não ser mais nenhuma novidade, o interesse por essa área da computação continua crescendo. Em meados da década de 90, com o surgimento do Beowulf [4] - o primeiro cluster contruído a partir de hardware e software convencionais - o acesso a eles tornou-se mais fácil. Isso ocasionou um maior esforço de pesquisa nessa área, visando a melhor utilização de um recurso agora cada vez mais acessível. Clusters deste tipo se tornaram bastante populares no meio acadêmico, principalmente por possuírem uma relação *custo x benefício* bem mais vantajosa do que a dos supercomputadores [17], tornado-se uma excelente alternativa para alcançar alto desempenho, mas a custos significativamente menores. Sua evolução têm acontecido rapidamente, e hoje existem clusters com mais de 128 mil nós [1]. Mais recentemente, a computação distribuída deixou de ser limitada por redes locais, e o conceito foi extrapolado para *grids* [3] e *nuvens* [2].

Clusters provêem uma infraestrutura que compartilha memória e processamento, oferecendo flexibilidade em termos de programação. Os dois principais paradigmas para programação em clusters são baseados na comunicação por troca de mensagens ou através de memória compartilhada distribuída

No modelo de programação por troca de mensagens, o programador tem controle total sobre a transferência dos dados. É responsabilidade dele decidir quando enviar os dados, pra quem enviar e o que enviar. Apesar desse controle dar ao programador a possibilidade de alcance do desempenho ótimo, isso torna a programação

nesse modelo complexa e de difícil depuração, especialmente se a aplicação utilizar estruturas de dados mais elaboradas. MPI [18] e PVM [19] são bibliotecas de código aberto para troca de mensagens entre processos em um ambiente de memória distribuída, e MPI acabou tornando-se o padrão para este paradigma.

No modelo de programação de memória compartilhada distribuída - DSM, *Distributed Shared Memory* - é criada uma abstração de memória compartilhada em cima de primitivas de envio/recepção de mensagens, e um espaço de endereçamento compartilhado é fornecido para o programador. Assim, uma aplicação pode ser escrita como se fosse executar em uma única máquina com vários processadores compartilhando a mesma memória, utilizando-se de operações simples de leitura e escrita. Toda a troca de mensagens e transferência de dados fica escondida do programador e é deixada a cargo do sistema DSM. Uma das grandes vantagens desse modelo é prover uma extensão natural do modelo de programação sequencial. Por outro lado, dois ou mais nós podem acessar os mesmos dados ao mesmo tempo, e eventualmente realizar uma operação de escrita sobre esses dados. Se acessos concorrentes não forem cuidadosamente controlados, eles podem ser executados em uma ordem diferente daquela esperada pelo programador. É necessário manter a *consistência* dos dados compartilhados pelo sistema. Mecanismos usados para a manutenção da consistência dos dados serão vistos em detalhes no Capítulo 2.

Em sistemas que executam em clusters, normalmente os problemas são resolvidos buscando-se apenas o aproveitamento de CPU, utilizando os nós envolvidos para realizar o processamento em paralelo de diversas partes das aplicações. Os dados utilizados são replicados em todos os nós, limitando assim a quantidade de memória do sistema àquela disponível localmente. Aplicações que utilizam muita memória, da ordem de dezenas de Gigabytes ou mesmo Terabytes, não conseguem se beneficiar de tais sistemas. Elas devem recorrer a algoritmos *out-of-core*, também conhecidos como algoritmos de *memória externa* [9], por realizarem acessos à memória secundária juntamente com a memória principal. Esses algoritmos buscam realizar E/S de forma eficiente, minimizando acessos ao dispositivo de armazenamento secundário - um disco rígido, por exemplo. Como os dados não cabem na memória disponível, a computação deve ser realizada em partes. Parte dos dados precisa ser lida para a memória para poderem ser usados. Terminada a computação, os dados

voltam para o disco para que outra massa de dados possa ser carregada na memória.

Outro ponto a ser observado é que, ao programar uma aplicação para executar em um cluster, as ferramentas utilizadas normalmente executam no nível de usuário. Elas costumam ser apresentadas na forma de compiladores [5] [6] ou de bibliotecas de linguagem [7] [8] [18] [19]. Independente da forma como são apresentadas, isso acaba adicionando uma camada de processamento entre a aplicação e o sistema operacional, aumentando ainda mais o overhead de comunicação entre eles. Além disso, os sistemas operacionais utilizados normalmente não são adaptados para esse tipo de sistema. Sistemas operacionais de propósito geral são projetados para uma ambiente computacional diferente daquele operado pelos clusters, e não possuem funcionalidades que permitam aproveitar toda a memória e a capacidade de processamento oferecidos. Dessa forma, cada nó, ao executar sua sua cópia local do S.O., acaba trabalhando de forma independente.

Nessa dissertação, é proposta uma solução para os problemas anteriormente citados. Foi criado um sistema capaz de oferecer para uma aplicação toda a memória disponível globalmente no cluster. Assim, é possível executar um programa que requeira mais memória do que a fisicamente disponível em um só nó, utilizando-se das memórias dos outros nós do sistema. O sistema operacional Linux 2.6 foi usado para a implementação de um protótipo, permitindo sua modificação dinamicamente através de módulos do kernel. O sistema de memória virtual foi modificado, de modo que ao sofrer uma falha de página, essa página seria buscada na memória de outro nó e não no disco. Essa integração do sistema proposto com o sistema operacional ajuda a diminuir a latência da comunicação entre aplicação e S.O. ao reduzir o número de trocas de contexto.

O sistema desenvolvido pode ser utilizado para a resolução de aplicações *out-of-core*. Como este tipo de aplicação lida com quantidades de dados maiores do que a memória principal de um computador, é necessário um número grande de acessos ao disco. Partindo do pressuposto que acessos à memória de um nó remoto, mesmo com a latência da rede, são mais rápidos do que os acessos à um dispositivo de armazenamento secundário, é esperado que essa classe de aplicações se beneficie do sistema proposto.

Foi disponibilizada para o programador uma biblioteca com funções para alocar

o espaço de endereçamento compartilhado e para inicializar a memória global no cluster, implementada no sistema através de um *pool* de memória distribuído. Para não aproveitar apenas a memória do cluster e deixar os processadores disponíveis inativos, foram adicionadas à biblioteca de programação primitivas de sincronização como barreiras e locks, para oferecer ao programador ferramentas para manter a coerência dos dados durante o processamento de aplicações paralelas.

1.2 Contribuições

Em resumo, as principais contribuições presentes nesta dissertação são:

1. A proposta de um modelo em *kernel* capaz agregar a memória dos nós de um cluster, possibilitando a execução eficiente de aplicações que utilizem mais memória do que a localmente disponível em um nó;

2. A implementação e avaliação desse mecanismo de agregação e compartilhamento de memória;

3. A disponibilização de uma biblioteca de programação com primitivas de sincronização, oferecendo ao programador meios para de manter a coerência dos dados em uma aplicação paralela.

1.3 Organização da Dissertação

Esta dissertação apresenta no Capítulo 2 os conceitos básicos de memória compartilhada e do funcionamento da memória virtual no Linux. No Capítulo 3 descrevemos o sistema e os detalhes de sua implementação. O Capítulo 4 descreve e analisa os experimentos utilizados para avaliar a nossa solução. Os trabalhos relacionados com o conteúdo desta dissertação encontram-se no Capítulo 5. O Capítulo 6 resume os principais resultados obtidos, apresenta as conclusões da dissertação e propõe alguns trabalhos futuros.

Capítulo 2

Conhecimentos Básicos

Nesse capítulo serão explicados alguns conceitos importantes para uma melhor compreensão da dissertação apresentada. O capítulo foi dividido em 2 assuntos principais: conhecimentos básicos de memória compartilhada distribuída em software e funcionamento da memória virtual no Linux.

2.1 Memória Compartilhada Distribuída em Software

Para a implementação do protótipo do sistema proposto, foram utilizados alguns mecanismos e algoritmos comuns em sistemas de memória compartilhada e distribuída em software (em inglês, *Software Distributed Shared Memory - SDSM*). Nesses sistemas, é oferecido um espaço de endereçamento virtual para os nós de um cluster. Esses endereços são utilizados pelos nós apesar de, fisicamente, a memória ser local a cada um deles. O espaço de endereçamento provê uma abstração onde toda a troca de mensagens envolvida no envio e recebimento de dados é escondida do programador.

Como vários nós podem acessar e atualizar ao mesmo tempo dados residentes no mesmo endereço, medidas devem ser tomadas para garantir que os dados compartilhados do sistema estejam sempre consistentes. Alguns mecanismos usados em sistemas SDSM e modelos de consistência de memória serão revisados no restante do capítulo.

A Figura 2.1 ilustra o funcionamento de um sistema SDSM. O sistema SDSM é

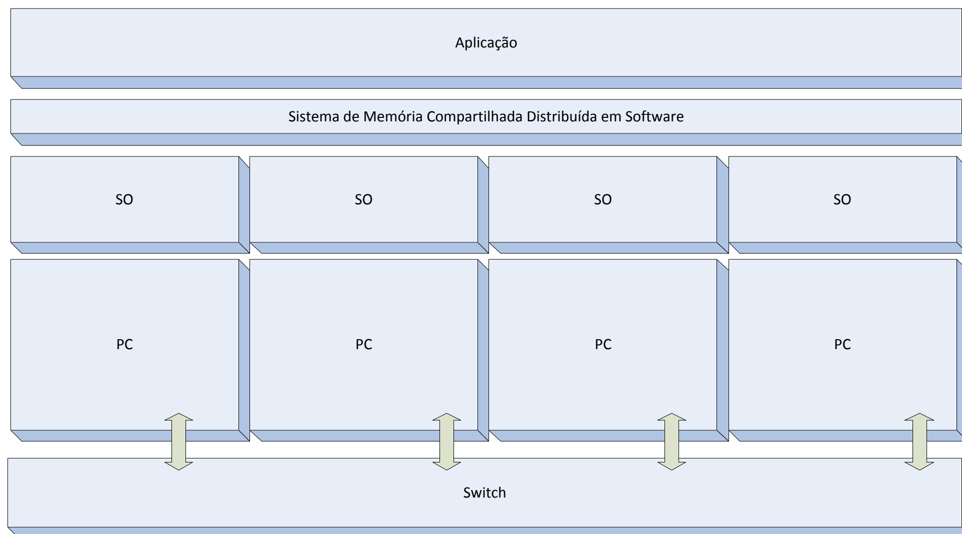

Figura 2.1: Esquema de funcionamento de um SDSM.

o responsável pela *interface* entre a aplicação e o sistema operacional, escondendo a complexidade da troca de mensagens do programador. Como pode-se notar na figura, os computadores envolvidos são interligados através de um *switch* e cada um possui sua própria cópia do sistema operacional.

2.1.1 Gerenciamento de sistemas SDSM

O modelo de gerenciamento mais simples e intuitivo é o centralizado, onde apenas um nó é o responsável pela gerência dos dados do sistema, administrando todos os pedidos de acesso aos dados. Nota-se, porém, que essa solução, apesar da simplicidade e facilidade de implementação, possui uma grande desvantagem: o nó gerente pode tornar-se rapidamente um gargalo, principalmente quando se aumenta o número de nós, pois não é *escalável*.

Um dos principais objetivos de desenvolver um gerente de memória distribuído é evitar a centralização do mecanismo de gerência de memória. No gerente de memória distribuído, o espaço global de memória compartilhada é dividido em segmentos de

mesmo tamanho. Esta divisão é realizada de acordo com o número de máquinas que farão parte da execução da aplicação. Cada máquina passa a gerenciar uma parte da memória global. Há, contudo, um aumento no overhead para gerenciar os dados do sistema. Como o gerenciamento da memória se torna mais compexo, o custo para localizar um bloco de dados ou o nó responsável por eles também aumenta.

2.1.2 Algoritmos SDSM

Algoritmos utilizados em sistemas SDSM [12] lidam com dois problemas básicos:

- a distribuição dos dados compartilhados através do sistema, visando minimizar a latência, e;
- a preservação de uma visão coerente dos dados compartilhados, minimizando a sobrecarga imposta pelo gerenciamento desta coerência.

Estes algoritmos são classificados nas seguintes categorias:

Algoritmos SRSW(Single Reader, Single Writer)

Esta é classe mais simples de algoritmos. Nesta classe, a replicação de dados não é permitida, e apenas um nó pode executar operações de leitura ou escrita sobre um determinado dado por vez. Desse modo, não há manutenção de mais de uma cópia de dados no sistema, e como consequência, o problema de consistência não existe. A migração de dados geralmente é usada, mas não é requerida. Esse tipo de algoritmo possui duas desvantagens principais: o efeito ping-pong e o falso compartilhamento [11]. O *efeito ping-pong* ocorre quando dois ou mais processadores estão acessando os mesmos dados, e como o acesso é feito de forma exclusiva, os dados ficam se movendo pela rede de um processador a outro. No *falso compartilhamento*, um ou mais processadores são bloqueados ao acessar o mesmo bloco de dados - uma página, por exemplo - mesmo que eles acessem posições de memória diferentes no bloco. Quanto maior é a granularidade dos dados, maior é o risco de sofrer falso compartilhamento.

Algoritmos MRSW(Multiple Reader, Single Writer)

Nesta classe de algoritmos, busca-se reduzir o custo das operações de leitura, partindo do pressuposto que leituras são o tipo de acesso mais comum em aplicações - sejam elas paralelas ou não. Desse modo, em um acesso de leitura os dados são replicados, para que vários nós possam realizar tal acesso sobre os mesmos dados simultaneamente. A replicação aumenta o desempenho do sistemas ao diminuir o custo da migração dos dados, pois o número de falhas de acesso também diminui. As operações de escrita, porém, são mais custosas, pois deve ser feita a manutenção da coerência das cópias, o que aumenta o fluxo de mensagens. Existem duas estratégias básicas para isso:

- Atualização - uma escrita em um dado compartilhado causa a atualização automática de todas as suas cópias. Essa abordagem procura diminuir o número de falhas de acesso à memória, mas em contrapartida gera um número alto de mensagens. Como na maioria das vezes uma atualização não é vista antes que outras sejam feitas, essa estratégia acaba enviando atualizações desnecessárias.
- Invalidação - uma escrita em um dado compartilhado causa a invalidação de todas as suas cópias, tornando-as inacessíveis. Um novo acesso a esses dados gera uma nova falha de acesso. A partir dessa falha, os dados atualizados são recuperados, fazendo com que mensagens de atualização sejam enviadas apenas sob demanda. Além disso, mensagens de invalidação são menores do que as de atualização, pois enviam apenas uma notificação ao invés de dados. Por esses motivos, o protocolo de invalidação é o mais utilizado.

Para operações de escrita, estes algoritmos também sofrem de falso compartilhamento e do efeito ping-pong.

Algoritmos MRMW(Multiple Reader, Multiple Writer)

Algoritmos que permitem múltiplos escritores [23] buscam reduzir o custo das operações de escrita, permitindo que dois ou mais processadores modifiquem concorrentemente um mesmo bloco de dados, e posteriormente combinem o resultado das modificações em pontos de sincronização, como locks ou barreiras. Múltiplos escritores ajudam também a reduzir os efeitos do falso compartilhamento. Softwares

DSM que utilizam esta técnica, como o Treadmarks [7], a implementam através de mecanismos de *twinning* e *diffing*. No Treadmarks, durante a realização de uma operação de escrita, é gerada uma violação de acesso. Isso ocorre porque, inicialmente, todas as páginas do sistema são *read-only*. A rotina de tratamento de violação de acesso faz então uma cópia da página - um *twin* - e libera a original para uso. Modificações são feitas apenas na cópia. Quando é necessario enviar as modificações para outros nós, a cópia e a versão atual da página são comparadas para ver quais são as modificações, e é gerado um *diff*. Somente ao chegar em pontos de sincronização é que os outros processadores são comunicados dessa alteração, através de *notificações de escrita*. Nos pontos de sincronização é necessário consultar uma lista de notificações de escrita para descobrir quais diffs precisam ser aplicados. Os diffs são solicitados e ao serem recebidos são aplicados à cópia desatualizada da página. Como os diffs contêm apenas as modificações realizadas na página, seu tamanho varia de acordo com a quantidade de dados modificados, mas normalmente essa quantidade é bem menor do que uma página.

2.1.3 Modelos de Consistência de Memória

Consistência de memória é a política que determina quando mudanças feitas por um processador são vistas pelos outros processadores do sistema. Um modelo de consistência define como o sistema DSM se comporta com relação as operações de leitura e escrita. Diferentes aplicações paralelas exigem diferentes modelos de consistência. Escolher o modelo certo para cada aplicação pode influenciar significativamente no desempenho do sistema. Modelos de consistência mais rígidos tipicamente aumentam a latência de acesso a memória e o fluxo de mensagens, enquanto simplificam a programação. Por outro lado, os modelos mais relaxados permitem reordenação de memória e realizam um número menor de atualizações, o que implica em aumento do desempenho. Entretanto, exigem maior envolvimento do programador na sincronização dos acessos aos dados compartilhados.

Consistência Seqüencial

O modelo de consistência de memória mais simples e intuitivo é chamado de consistência seqüencial. Este modelo assegura que todos os nós vêem a mesma seqüência

de leituras e escritas. Para que um programa seja sequencialmente consistente, é preciso que todos os acessos de memória sejam feitos na ordem especificada no programa. Sistemas que obedecem a essa condição são fortemente ordenados, o que tende a diminuir seu desempenho.

Consistência de Processador

Esse modelo assume que a ordem na qual diferentes processadores podem ver operações de memória não precisa ser idêntica, mas todos os processadores devem observar a sequência de escritas demandada por cada processador, e na mesma ordem. Comparado com a consistência sequencial, esse modelo consegue um ganho de desempenho por tratar leituras de forma diferente de escritas.

Consistência Fraca

Requer que a memória se torne consistente somente em acessos de sincronização. Um acesso de sincronização deve esperar por até que todos os acessos anteriores sejam completados, enquanto operações normais de leitura e escrita devem esperar somente que acessos de sincronização anteriores sejam completados. Acessos de sincronização são garantidos como seqüencialmente consistentes. É responsabilidade do programador a consistência dos dados compartilhados através do uso correto de operações de sincronização.

Consistência Relaxada

É uma extensão do modelo de consistência fraca, onde acessos de sincronização são divididos em operações de obtenção (*acquire*) e liberação (*release*). Uma obtenção indica que o processador está iniciando uma operação que pode depender de valores gerados por outro processador. A execução de uma liberação indica que o processador está terminando uma operação que gerou valores dos quais outros processadores podem depender. Um modelo de consistência relaxada [13] pode ser ansioso (*eager*), quando propaga as modificações na memória compartilhada em cada liberação, ou preguiçoso (*lazy*), quando as modificações esperam até que uma próxima operação de obtenção.

2.2 O Sistema de Gerenciamento de Memória Virtual no Linux

2.2.1 Paginação e Memória Virtual

O *kernel* do Linux trata páginas físicas da memória principal como a unidade básica do gerenciamento de memória. Apesar da menor unidade de endereçamento do processador ser normalmente uma palavra, a MMU (*Memory Management Unit*, o hardware que gerencia memória e traduz endereços virtuais para físicos) tipicamente lida com páginas [10]. Para arquiteturas de 32 bits, elas normalmente tem 4096 bytes. O kernel representa toda página física no sistema com uma estrutura do tipo `struct_page`. Essa estrutura possui, entre outros, um campo que aponta para o endereço virtual da página. Como um endereço virtual pode ser mapeado para qualquer endereço físico, páginas podem ser movidas da memória principal para o disco, e depois de volta para a memória principal, mantendo o mesmo endereço virtual. Através desse mecanismo que é feita a implementação de memória virtual em sistemas operacionais modernos: o disco é usado como uma extensão da memória RAM, mas de maneira transparente para os processos.

2.2.2 Espaço de Endereçamento de um Processo e Regiões de Memória

O *espaço de endereçamento* de um processo consiste do intervalo de todos os endereços virtuais que um processo pode acessar. Cada processo possui seu próprio espaço de endereçamento, que é independente do espaço de endereçamento de outros processos [14].

Toda a informação relacionada ao espaço de endereçamento de um processo está contida no descritor de memória desse processo, que é implementado pelo kernel do Linux através de uma estrutura do tipo `mm_struct`. Esta estrutura é referenciada diretamente pelo processo através do seu descritor, que é representado pela estrutura (`task_struct`).

Um espaço de endereçamento é composto de várias *regiões de memória* (ou VMAs, *Virtual Memory Areas*), que é a abstração utilizada pelo kernel do Linux

para representar intervalos de endereços virtuais. Regiões de memória são caracterizadas por um endereço inicial, o tamanho da região e suas permissões de acesso, e identificam um intervalo contínuo e homogêneo de endereços virtuais. O endereço e o tamanho devem ser múltiplos de 4096 para que a região possa preencher completamente todas as páginas alocadas.

A Figura 2.2 ilustra melhor o encadeamento dessas estruturas no kernel. O descritor de memória de um processo aponta para sua lista de VMAs. Cada VMA, por sua vez, define um intervalo de endereços virtuais, que são os endereços acessados pelo processo.

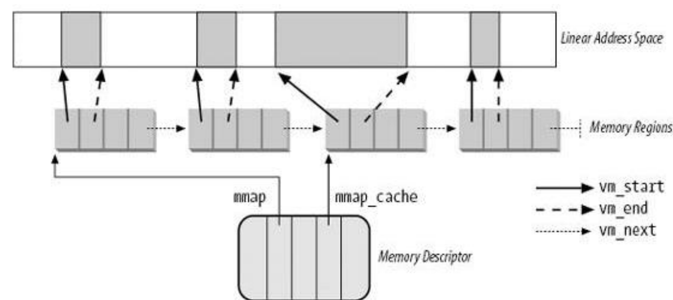

Figura 2.2: Regiões de memória e o espaço de endereçamento de um processo.

Uma VMA é representada através da estrutura `vm_area_atruct`. Nessa estrutura, o campo `vm_ops` aponta para um conjunto de operações associadas com a região de memória, que podem ser invocadas pelo kernel para manipular a VMA [15]. A operação mais importante desse conjunto é a `nopage()`. Essa função é usada pela rotina de tratamento de falhas de página da seguinte maneira: se ela estiver definida para a VMA em questão, ela é invocada e executada durante uma falha de página nessa região. Assim, durante o tratamento dessa falha, as páginas receberão o tratamento específico definido na função `nopage()` para essa VMA.

Capítulo 3

Implementação de Memória Global no Kernel

3.1 Projeto

O sistema de memória global no kernel foi projetado visando agregar a memória dos vários nós de um cluster, e não apenas compartilhá-la. Isto é feito utilizando a memória virtual para mapear o espaço de endereçamento de uma aplicação nos outros nós computacionais de um cluster.

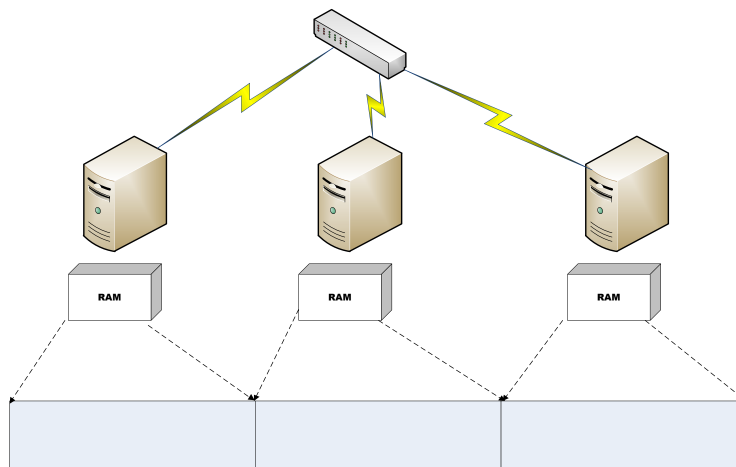

Figura 3.1: Espaço de endereçamento compartilhado.

O protótipo foi implementado no kernel 2.6.18.8 do Linux. A versão do kernel

utilizada foi a de 32 bits, compilada para suporte total ao PAE (*Physical Address Extension*). Com o PAE, o espaço de endereçamento passa de 32 para 36 bits, aumentando o tamanho máximo da memória que pode ser endereçada de 4 para 64 gigabytes.

Linux é um sistema operacional de código aberto, que disponibiliza seu código fonte sob a licença GPL [16], além de oferecer o uso de módulos em seu kernel. Módulos são partes do kernel que podem ser carregadas ou descarregadas dinamicamente, adicionando funcionalidades ao kernel residente. A implementação em modo kernel visa minimizar o número de trocas de contexto e integrar a solução ao sistema de memória virtual do Linux, visando melhor performance para o sistema, como pode-se perceber na Figura 3.2. A aplicação se comunica diretamente com o sistema operacional, eliminando o *overhead* de comunicação imposto por sistemas SDSM que executam no nível de usuário, como visto na Figura 2.1. Modificações no kernel residente foram necessárias e serão explicadas em detalhes na próxima seção, mas a maior parte do sistema foi escrita em módulos do kernel.

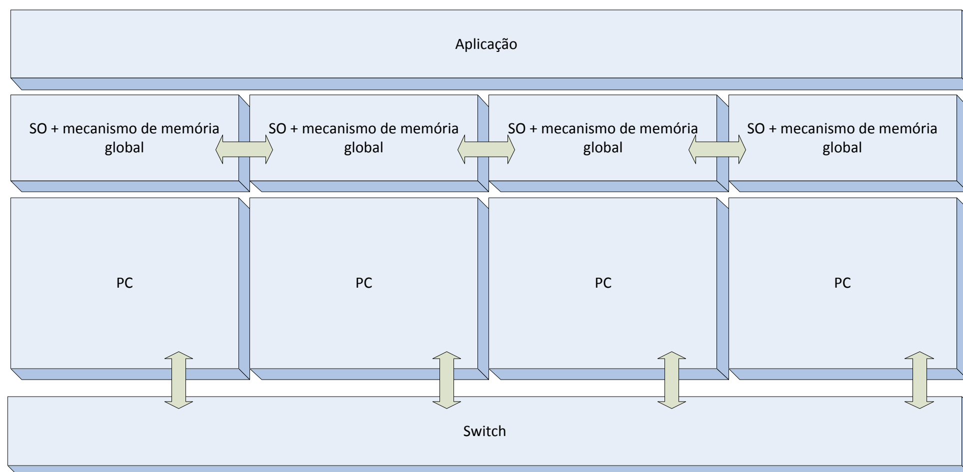

Figura 3.2: Integração do sistema com o S.O.

No gerenciamento da memória é utilizado um protocolo baseado em residências [21]. As páginas inicialmente são divididas estaticamente entre os nós, e um nó se torna a residência - o *home* - de sua fração de páginas, isto é, torna-se o responsável pelo gerenciamento destas. Todas as páginas são unicamente identificadas no sistema através de seu PID (*Page ID*). Inicialmente, um nó só possui acesso às páginas das quais ele é o *home*. Ao acessar páginas de outros nós *home*, uma falha de página é gerada e ela será buscada em seu nó *home*. O PID e o ID da sua residência são calculados e então uma mensagem de requisição de página será enviada ao nó *home*, realizando assim a troca de dados entre dois nós do sistema.

O sistema pode executar duas classes de aplicações - sequencial e paralela - e sempre utiliza dois ou mais nós em suas execuções. Para aplicações sequenciais, apenas um nó realiza computação enquanto os restantes são reponsáveis por responder as requisições de páginas feitas por esse nó, isto é, há apenas compartilhamento de memória, e não de processamento. Como apenas um nó realiza computação e não existem acessos concorrentes, todos os acessos são tratados como escrita. Na prática, o maior impacto de se tratar os acessos deste modo ocorre quando a memória local do nó que realiza computação fica cheia. Quando isto acontece, deve ser liberado espaço na memória para que novas páginas possam ser alocadas. Para liberar memória, as páginas devem ser enviadas de volta para suas residências - como os acessos são tratados como escrita, o sistema assume que houve modificação nas páginas, mesmo que na verdade elas tenham apenas sido lidas.

Em aplicações paralelas, todos os nós envolvidos realizam computação. Para esta segunda classe de aplicações, o sistema é implementado através de um algoritmo *MRSW*, usando *invalidação de dados* combinado com um protocolo de consistência relaxada ansiosa. Inicialmente, todos os acessos são tratados como leitura. Desse modo, quando a memória local de um nó enche e é preciso liberar espaço, as páginas podem ser simplesmente liberadas - para acessos de leitura não há modificação nas páginas, então elas não precisam ser enviadas de volta para a sua residência. Para realizar a atualização de dados em aplicações paralelas de forma correta (propagando as mudanças para o nó *home*), operações de escrita devem ser realizadas entre duas primitivas de sincronização, `lock_acquire()` e `lock_release()`. O funcionamento do sistema será explicado em detalhes nas próximas seções.

3.2 Implementação

3.2.1 Dificuldades encontradas durante a implementação

A maior dificuldade encontrada ao realizar a implementação do protótipo foi a de programar no kernel. Os *releases* do kernel mudam com frequência [25], e não há documentação que acompanhe estas mudanças. Existem também várias limitações em comparação com a programação em nível de usuário, em sua maioria visando melhor performance em detrimento de legibilidade de código. Mensagens de erro também não são amigáveis, retornando o conteúdo de registradores e a árvore de chamadas até a função com erro, normalmente antes de travar a máquina ou reiniciá-la, deixando o desenvolvimento ainda mais lento. Isso acabou impossibilitando a implementação e alguns testes de serem realizados à distância (via ssh ao cluster), pois qualquer travamento requeria uma pessoa para reiniciar a máquina através do botão de reset.

3.2.2 Modificações no kernel

O sistema foi desenvolvido, em sua maior parte, em módulos do kernel. Módulos podem ser carregados e descarregados dinamicamente, e melhor, podem ser modificados sem que seja necessário recompilar o kernel residente. Algumas mudanças, no entanto, foram necessárias no kernel residente. Algumas funções, que originalmente não são visíveis à módulos, tiveram que ser exportadas. As funções são as seguintes:

- `make_pages_present`: força uma falha de página para cada página de um determinado intervalo, trazendo-as para memória. É utilizada durante a inicialização do sistema, para o intervalo de páginas que o nó é responsável por gerenciar;

- `zap_page_range`: desfaz o mapeamento de uma página na memória e libera todas as estruturas relacionadas. Usada no encerramento do sistema, para dissociar uma página (`struct page`) de um endereço e liberar essa estrutura.

Foi ainda adicionada ao kernel a função `kgm_zap_page_range`, uma modificação da função `zap_page_range`, que desfaz o mapeamento mas não libera suas estruturas.

Essa função é necessária para casos onde é preciso apenas desfazer um mapeamento, de modo que novas falhas de página possam ocorrer naquele endereço, mas sem descartar a estrutura associada, já que ela continuará sendo utilizada no sistema.

Por fim foram criadas cinco chamadas de sistema para fazer a comunicação entre o kernel e a aplicação. A partir da versão 2.6 do kernel, a tabela de chamadas de sistema passou a ser protegida para escrita, além de não ser mais exportada pelo kernel, ou seja, não ser visível aos módulos. Essa modificação aconteceu porque, com a facilidade de acesso à tabela, a interceptação de chamadas se tornava trivial, o que facilitava a criação de códigos maliciosos. Assim não há mais como criar as chamadas através apenas de módulos, e modificações no kernel se fazem necessárias. Implementá-las apenas no kernel também não é desejável, já que qualquer modificação demandaria nova compilação do kernel. As chamadas foram criadas no kernel residente, mas fazendo com que o seus códigos fonte possuam apenas ponteiros para funções, inicialmente apontando para `NULL`. Os ponteiro são então exportados, de forma que um módulo possa atualizá-los e apontá-los para funções válidas. Assim, apenas o módulo precisa ser recompilado ao se modificar uma das chamadas de sistema criadas. As chamadas serão descritas a seguir:

- `kgm_mmap`: Aloca um espaço de endereçamento e realiza a atualização da estrutura `vm_operations` da VMA com um ponteiro para a função `nopage()` customizada;
- `kgm_init`: Inicializa as estruturas utilizadas para gerenciamento do sistema e conecta os nós através de sockets TCP;
- `kgm_acquire`: Realiza as operações necessárias para manter a coerência de dados ao conseguir acesso a um *lock* e entrar em uma região crítica;
- `kgm_release`: Encerra o acesso a uma região crítica e libera o *lock*;
- `kgm_exit`: Libera todas as estruturas, *sockets*, *threads* e regiões de memória criadas.

3.2.3 Preparação do sistema

Antes de começar a executar aplicações, é necessário executar o script `globalmem.sh`, que irá passar para o sistema informações a respeito dos nós participantes. Foram testados vários modos de trocar essas informações entre o kernel e o espaço de usuário. Além das chamadas de sistema, foi também cogitado o uso de *netlink sockets*. Estes *sockets* são usados como mecanismos para IPC entre o kernel e processos de usuário, oferecendo um *link* de comunicação bidirecional entre eles. Contudo, a criação de arquivos no diretório `/sys`, que pode ser acessado tanto pelo kernel quanto pelo espaço de usuário, mostrou-se mais simples de implementar e menos invasiva ao kernel, e por esses motivos foi o método escolhido. O script tem como argumento um arquivo texto com os endereços IP dos nós, e a partir dessas informações calcula o número de nós participantes e o seu ID no sistema, passando essas informações para os módulos que são carregados durante a execução do script antes da execução das aplicações. O script executa da seguinte forma:

1. São criados os arquivos `num_hosts` (número de nós), `hosts` (lista com os endereços dos nós) e `whoami` (ID do nó no sistema) em `/sys/globalmem`;

2. Os arquivos criados são preenchidos a partir das informações contidas no arquivo de endereços IP passado ao script, e desse modo essas informações podem ser acessadas a partir do kernel;

3. Os arquivos são lidos no kernel e suas informações passadas a estruturas internas. Nesse ponto o nó já sabe quantos e quais nós participarão da computação e qual é o seu ID no sistema;

4. São acertados os ponteiros das chamadas de sistema criadas de modo que elas enxerguem as funções presentes nos módulos carregados.

3.2.4 Inicialização do sistema

Após o sistema estar preparado, com informações a respeito dos nós participantes e as chamadas de sistema apontando para funções válidas, um programa que utilize a biblioteca disponibilizada já pode ser executado. A seguir será descrito o processo de inicialização dessas aplicações.

Memória

Ao inicializar o sistema, deve-se primeiro alocar o espaço de endereçamento que será compartilhado pelos nós. A chamada de sistema `kgm_mmap()` faz esse trabalho através da função `do_mmap_pgoff`. Um espaço de endereçamento é retornado, e a VMA associada a esse intervalo é descoberta com a função `find_vma`. Essa VMA será gerenciada pelo sistema, e para isso é necessário atualizar o ponteiro da estrutura `vm_operations` associada à ela. Assim, sempre que ocorrer uma falha de página em um endereço válido na VMA, ela receberá tratamento específico pela rotina definida no protótipo.

Após alocar o espaco de endereçamento é que de fato ocorre a inicialização das estruturas do sistema, através da chamada `kgm_init()`. Primeiramente, com base no número de páginas e nós no sistema, é calculado o intervalo de páginas pelas quais o nó será a residência. Uma estrutura chamada `kgm_pg_mngr` é criada para auxiliar o gerenciamento dessas páginas, mantendo, para cada página, uma lista com os nós que possuem uma cópia sua.

Uma área de memória é alocada e reservada para uso exclusivo do sistema, funcionando como um *pool* de páginas. Todas essas páginas são reservadas, fazendo com que o sistema operacional não possa mais substituí-las. Assim têm-se a garantia de que elas estarão sempre na memória. Tentou-se utilizar a *flag* `VM_LOCKED`, durante a alocação da VMA, para isso. Contudo, todas as páginas do intervalo eram marcadas como *não substituíveis*, o que na prática impedia a ocorrência de falhas de páginas ao acessar qualquer endereço da VMA. Como o comportamento desejado era reservar apenas as páginas da cache, foi utilizada a função `SetPageReserved`.

As páginas do *pool* estão organizadas da seguinte forma: as páginas livres, que são aquelas disponíveis para uso, são organizadas em uma lista. As páginas home (páginas que o nó é a residência) e as páginas não home (páginas que pertencem a outros nós, mas o nó possui uma cópia localmente) - são organizadas em uma *árvore AVL*, ordenadas pelo ID das páginas. Essa estrutura de dados é uma árvore binária de busca auto-balanceada. Nessa árvore, as alturas das duas sub-árvores a partir de cada nó difere no máximo em uma unidade. As operações de busca, inserção e remoção de elementos possuem complexidade $O(\log n)$, onde n é o número de elementos da árvore. É utilizada a versão iterativa desse algoritmo, pois recursão

é fortemente desencorajada no kernel. Em testes iniciais no protótipo, as páginas *home* e as *não-home* eram organizadas também em listas, assim como as páginas livres. Testes realizados, porém, revelaram o péssimo desempenho de operações de busca nessa estrutura. A fim de melhorar o desempenho do sistema, estas estruturas tiveram que ser melhoradas.

Ao ocorrer uma falha de página no sistema, primeiro uma página é alocada no *pool* do sistema. Esta página é então retirada da lista de páginas livres e adicionada à estrutura de páginas *home* ou à estrutura de páginas *não-home*.

Se o *pool* ficar sem páginas livres, é invocada a função `kgm_oom()` para liberar páginas. Essa função pode ser ajustada com a porcentagem de páginas que devem ser liberadas - atualmente, o sistema libera todas as páginas. Caso uma página que esteja sendo acessada para leitura seja liberada, simplesmente é desfeito o mapeamento da página com o seu endereço associado e ela volta para a lista de páginas livres. Caso a página esteja sendo acessada para escrita, é preciso enviá-la de volta para o seu home - para que ele atualize suas informações - antes de colocar a página na lista de páginas livres.

Após o *pool* ser inicializado, é definido um intervalo de páginas para cada nó com base no número de nós e número de páginas total no sistema. É gerada uma falha de acesso para cada página desse intervalo com a função `make_pages_present`, para que o nó as tenha em memória. Assim o nó torna-se a residência dessas páginas e será responsável por todos os acessos a elas.

Nesse ponto, todas as estruturas internas já estão inicializadas. As estruturas usadas são as seguintes:

- `kgm_info`: Estrutura que contém informações sobre os nós participantes, como o ID do nó, o número total de nós e o IP dos nós.
- `kgm_host_mem`: Possui informações sobre a memória global e local do sistema, como o número e o intervalo de páginas home do nó, o número de páginas total, etc.
- `kgm_cache`: Usada para gerenciar o *pool* de páginas do sistema. Possui 3 listas - `free_pages`, com as páginas livres; `home_pages`, com as páginas home; e `used_pages`, com as páginas usadas mas que pertencem a outro nó. Possui

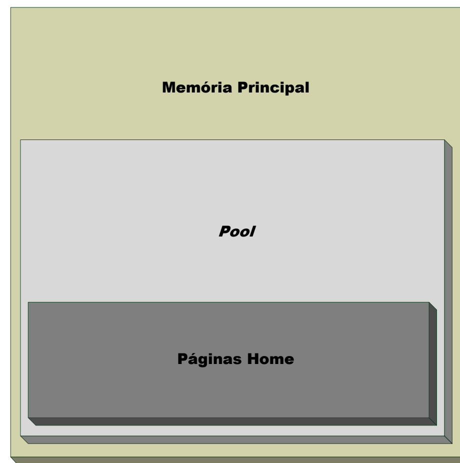

Figura 3.3: Organização da memória do sistema.

ainda contadores para os números de páginas livres e páginas não-home.

- `kgm_pg_mngr`: mantém informações necessárias para gerenciar as páginas home, como quais nós possuem uma cópia sua e se a página está em lock (sendo atualizada) ou disponível.

Comunicação

Definida a distribuição de páginas pelos nós e alocado o *pool* do sistema, é preciso criar e conectar os sockets utilizados para a comunicação entre os nós. São utilizados sockets TCP, tanto para mensagens de dados como de controle. Cada nó é conectado a todos os outros nós do sistemas através de três sockets: um socket que vai cuidar das mensagens de controle (`request_socket`), um socket para receber as páginas requisitadas pela rotina de tratamento de falhas de página (`reply_socket`) e um socket para receber as páginas que foram modificadas por outros nós (`dirt_socket`). Depois de criados, os sockets ficam escutando outras conexões.

Para realizar a conexão entre os nós, *kernel threads* são utilizadas. Em um mesmo nó, existe uma *thread* para cada outro nó no sistema, e cada uma é responsável por tratar as requisições de um determinado nó. Ao iniciar, uma *thread* se conecta com todos os outros nós remotos, através dos três sockets descritos anteriormente. Uma dificuldade encontrada na hora realizar a conexão dos sockets no kernel é que a

função `accept()`, ao contrário da sua versão na glibc (no espaço de usuário), não permite filtrar o endereço do socket que irá conectar. Assim, a função `accept()` no kernel aceita conexões provenientes de qualquer nó. Inicialmente, nas primeiras versões do protótipo, as kernel threads eram responsáveis por aceitar as conexões. Quando o número de nós era superior a dois, no entanto, um problema surgiu: muitas vezes, os sockets de dados e o socket de controle de um nó se conectavam a threads diferentes de um mesmo nó remoto. A solução foi forçar a conexão a partir das threads, para assegurar que, no seu lado da conexão, todos os sockets são associados ao mesmo endereço IP. No outro lado, eventualmente os sockets podem ser associados às estruturas erradas na hora do `accept()`. Como essas estruturas ficam concentradas no mesmo fluxo de execução do código, e não espalhadas por diferentes threads, basta percorrê-las verificando o IP do nó que está conectado e trocar os ponteiros dos sockets adequadamente.

Após conectadas, as threads podem receber as mensagens de controle através do `request_socket` e passam a tratar essas requisições.

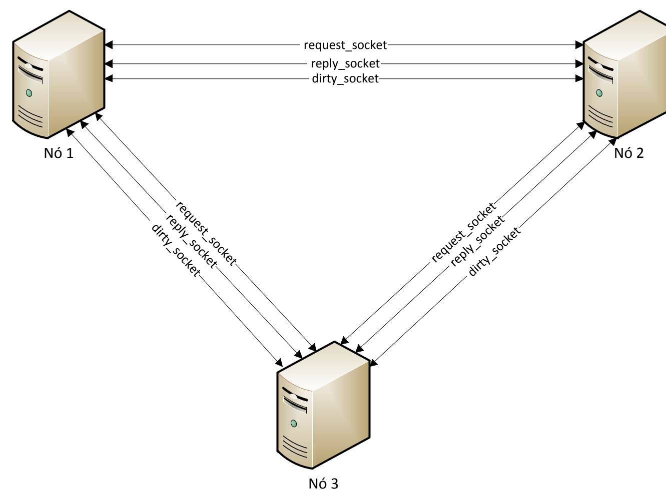

Figura 3.4: Esquema de conexão dos sockets.

Os tipos de mensagens enviadas são:

- `PAGE_REQUEST`: É uma mensagem de requisição de página. Quando ocorre uma falha de página, o nó envia um `PAGE_REQUEST` a outro nó, pedindo uma página com um determinado tipo de acesso.

- `PAGE_REPLY`: Mensagem que contém os dados de uma certa página do sistema. É enviada como resposta a um `PAGE_REQUEST`.

- `PAGE_INVALIDATE`: Mensagem de invalidação. Ao acessar uma página para escrita, o nó home da página consulta quem tem cópias da página e envia uma mensagem de `PAGE_INVALIDATE` a cada um desses nós, para avisá-los que a cópia que eles possuem vai ser escrita e, portanto, tornar-se desatualizada.

- `PAGE_DIRT`: Mensagem utilizada para enviar a cópia atualizada de uma página para o seu nó *home*, depois que ela foi alterada por outro nó.

- `KGM_ACK`: Usada para avisar um nó que uma mensagem do tipo `PAGE_DIRT` ou `PAGE_INVALIDATE` foi recebida e tratada, e outra mensagem, se houver, já pode ser enviada. Esse tipo de mensagem precisou ser criado pois as mensagens anteriormente citadas, por serem normalmente enviadas em rajadas para um mesmo nó, estavam truncando no buffer de recepção.

- `KGM_EXIT`: Essa mensagem sinaliza que a aplicação está sendo terminada e que as *threads*, *sockets* e demais estruturas podem ser liberadas.

3.2.5 Funcionamento do sistema

Uma vez inicializado o sistema, a computação pode começar. Durante a execução de uma aplicação, três mecanismos são os principais responsáveis pela troca de páginas entre os nós e a manutenção da coerência dos dados: o mecanismo de falha de página, as barreiras (que mesmo para aplicações sequenciais são usadas internamente pelo sistema) e, para aplicações paralelas, os *locks*. Através desses mecanismos o programa interage direta ou indiretamente com as páginas alocadas.

Falha de Página

Uma falha de página ocorre quando um processo precisa acessar os dados que se encontram em uma determinada página do sistema, mas a página não se encontra na memória do nó em que o processo está rodando. É necessário então localizar essa página e buscar uma cópia na memória de um nó remoto.

Ao ocorrer uma falha de página, primeiramente é calculado o PID (*page ID*) da página. Como as VMAs têm o mesmo tamanho, mas podem ser alocadas em endereços virtuais diferentes, o cálculo do PID é baseado no endereço inicial da VMA e no offset desse endereço até o que recebeu a falha. Através do PID da página, do número global de páginas no sistema e do número de nós, pode-se facilmente calcular o ID do nó home dessa página, já que a distribuição de páginas é feita estaticamente durante a inicialização. Descobertas essas informações, uma página livre é alocada do *pool* e uma mensagem de `PAGE_REQUEST` é enviada ao home da página, requisitando-a. O nó, ao receber essa mensagem, verifica o tipo de acesso pedido. Se for um acesso simples de leitura, uma cópia da página é enviada e o nó destino da página é marcado como possuidor de uma cópia dessa página. Se o acesso for de escrita, uma mensagem de invalidação deve ser enviada aos nós que possuem uma cópia da página, avisando-os que a página vai ser escrita e se tornar desatualizada.

Ao receber a página, é preciso aplicar os dados recebidos. Por estar dentro da rotina de tratamento de falha de página, não é possível simplesmente copiar os dados no endereço que sofreu a falha - a rotina ainda não retornou, então o endereço continua sem ter uma `struct page` associada a ele. Através das funções `kmap_atomic` e `kunmap_atomic` é possível criar um mapeamento temporário para a `struct page` alocada e então copiar o dados para esse mapeamento. Assim, a rotina de tratamento de falha de página retornará uma página já preenchida com dados para o usuário.

Barreira

Barreira é o mecanismo de sincronização mais simples em uma aplicação paralela. Ela é apenas um ponto no código onde um processo pára de executar e espera todos os outros processos também atingirem esse ponto. Só então a computação pode

voltar a acontecer. Isso garante que todos os processos chegaram no mesmo ponto do código.

A barreira é implementada no *userspace* apenas, não sendo necessária nenhuma consulta ou modificação à estruturas do kernel. O gerente da barreira é definido estaticamente como o nó com ID 0. Com a função `bar_init()`, o nó 0, gerente da barreira, dispara uma *thread* e espera os outros nós se conectarem a ela. Uma vez conectados, a barreira é inicializada. A barreira é definida em uma aplicação através da função `bar_wait()`. Os nós enviam uma mensagem à *thread* que gerencia a barreira e esperam uma resposta para prosseguir com a computação. Essa resposta só é enviada de volta quando todos os nós envolvidos atingem a barreira.

Lock

Os *locks* são primitivas de sincronização utilizadas para implementar *regiões críticas*. Desse modo, pode ser evitado o acesso simultâneo à váriávies compartilhadas e a coerência dos dados pode ser garantida.

Um ponto importante a ser reforçado antes de explicar o funcionamento do mecanismo de *lock* é que a implementação do sistema utiliza falhas de página, mas não falhas de proteção - que são geradas ao tentar realizar uma escrita em uma página marcada como *somente leitura*. A idéia inicial, ao realizar a implementação do sistema, era de utilizar estes dois tipos de falha. Tentativas de trabalhar com falha de proteção não deram certo devido a problemas de implementação. Por esse motivo tal mecanismo não foi utilizado, resumindo o sistema ao uso de falhas de página apenas. Na prática, isso implica que não há como diferenciar os acessos de leitura dos acessos de escrita para o sistema. Sempre que há uma falha de página, a página buscada pode ser tanto lida quanto escrita. Para aplicações sequenciais, todo acesso é interpretado como uma escrita, já que apenas um nó realiza computação, enquanto os outro apenas tratam requisições de páginas. Na versão paralela do sistema, acessos são tratados como leituras, embora não haja impedimento algum para escrever nessas páginas. Mas apenas escritas realizadas entre duas primitivas `lock_acquire()` e `lock_release()` são propagadas, mantendo a coerência dos dados. Desse modo, acessos realizados fora do *lock* são tratados como acessos de leitura, e acessos dentro do *lock* são tratados como escrita.

Ao executar uma instrução de `lock_acquire()`, um pedido de *acquire* é enviado ao gerente do *lock*, que é definido estaticamente como o nó com o ID mais alto. No gerente, o acesso ao *lock* é guardado por um *mutex*. Ao conseguir acesso, a *system call* `kgm_acquire()` é chamada e todas as páginas *não-home* são enviadas de volta para seus nós *home*. Isso deve ser feito porque, como explicado anteriormente, acessos realizados fora do lock são tratados como leitura. Assim, uma nova falha de acesso é necessária para que o acesso seja tratado corretamente como escrita. A operacão de *acquire* é terminada e as escritas podem ocorrer até que um seja chamada a instrucão de *release* para o *acquire* correspondente. Nos acessos de escrita, mensagens de invalidação são enviadas para cada nó que possui uma cópia da página sendo acessada. Assim, o nó fica sabendo que a página que ele possui está desatualizada.

No *release*, uma cópia de todas as páginas não home - aquelas que foram acessadas dentro do lock - é enviada de volta ao seu home para que ele tenha a versão mais atualizada da página. Uma mensagem então é enviada ao gerente do lock avisando que o *lock* está liberado.

3.2.6 Encerramento do sistema

Ao terminar toda a computação, a *system call* `kgm_exit()` deverá ser chamada pela aplicação para liberar todas as estruturas internas. Ao executar `kgm_exit()`, todas as páginas do *pool* são liberados para que o SO tenha novamente controle sobre elas. Em seguida, uma mensagem `KGM_EXIT` é enviada para todos os outros nós, para avisar que não irá mais realizar nenhum trabalho útil para eles. A *thread* que recebe a mensagem libera seus *sockets* e é então terminada, já que não terá mais utilidade para a aplicação. O nó, entretanto, precisa receber essa mesma mensagem dos outros nós. Quando recebe todas as `KGM_EXIT`s esperadas, os *sockets* restantes e as estruturas alocadas podem ser liberados. Por fim, o espaço de enderaçamento alocado também é liberado e a aplicação pode ser encerrada.

Capítulo 4

Análise Experimental

Este capítulo descreve os aspectos relacionados ao ambiente experimental e os testes efetuados para analisar o protótipo. Através destes foi feita uma análise dos resultados, explicando em detalhes o desempenho obtido.

4.1 Ambiente de testes

O ambiente experimental é composto por 4 máquinas, cuja especificação se encontra na tabela 4.1. As máquinas são interligadas por um *switch Gigabit Ethernet*.

Item	Descrição
Processador	Intel Xeon Quad Core E5410 2.33GHz com cache L2 de 12MB
Memória RAM	8GB
Interface de Rede	10/100/1000 Mbps
Sistema Operacional	Linux 2.6.18.8
Unidade de Disco	320 GB SATA 5400 RPM com cache de 32MB

Tabela 4.1: Configuração das Máquinas

4.2 Métricas de desempenho

Algumas métricas foram usadas na avaliação dos resultados obtidos com os testes do protótipo. Sua utilização permite uma melhor compreensão de como foi gasto o tempo total de *execução* das aplicações. As métricas são descritas a seguir:

- Tempo de barreira - mede o tempo gasto esperando outros nós em uma barreira.

- Tempo de lock - mede o tempo gasto nas operações de `lock_acquire()` e `lock_release()`, onde é realizada a manutenção da coerência dos dados.

- Tempo de falha de página - é a métrica mais importante, o principal ponto do sistema a ser analisado e otimizado. Quanto menor o tempo gasto tratando uma falha de página, maior a chance de se obter melhor desempenho. Foi dividida em 2 submétricas:

  - Tempo de Alocação: o tempo para alocar uma página no *pool* do sistema;
  - Tempo de Busca de Página: é o tempo de requisitar uma página à um nó remoto e recebê-la de volta.

4.3 Problemas encontrados durante os experimentos

Ao realizar os testes no protótipo, a fim de colher os resultados para a redação do presente capítulo, um problema foi encontrado: a função `do_mmap_pgoff`, utilizada para alocar o espaço de endereçamento a ser compartilhado, não retornava intervalos de endereços com tamanho maior do que 1Gb. Verificou-se que, na versão de 32 bits, o kernel só trabalha com 1Gb de memória, deixando o restante para o espaço de usuário. Essa divisão entre o kernel e o espaço de usuário não é fixa, podendo ser modificada. Porém, até a data de escrita deste capítulo, uma solução não foi encontrada para a versão do kernel utilizada (2.6.18.8). Por esse motivo, os espaços de endereçamento utilizados durante os testes que seguem são menores do que 1Gb.

4.4 Aplicações

Para testar o protótipo, foram utilizadas duas aplicações: *multiplicação de vetores* e *mergesort*. Elas serão descritas e analisadas a seguir.

4.4.1 Multiplicação de vetores

Nessa aplicação, é simplesmente realizada um multiplicação entre dois vetores. O elemento i de um vetor é multiplicado pelo elemento i do outro vetor, com o resultado armazenado no primeiro, ou seja, temos uma operação na forma *X[i] = X[i] * Y[i]*. Esta aplicação possui um padrão de acesso bem definido, o que facilita uma primeira avaliação do protótipo criado.

Resultados

Foram realizados dois experimentos com essa aplicação. Primeiramente é analisada sua execução sequencial, com apenas um nó realizando computação útil, enquanto os outros apenas respondem a requisições de páginas. O segundo experimento analisa a execução de uma versão paralela da aplicação, onde todos os nós realizam computação sobre um número igual de elementos dos vetores. São utilizados dois vetores de 476Mb cada, totalizando 952Mb de memória global compartilhada, ou 243.952 páginas.

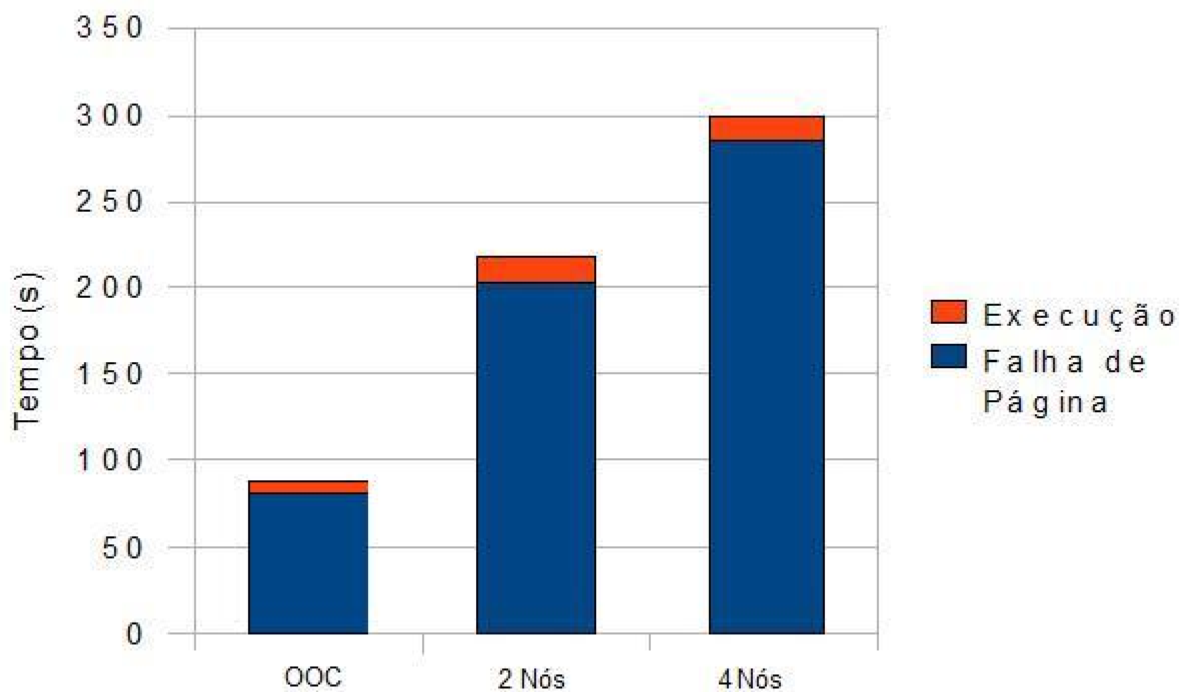

Figura 4.1: Tempo de execução sequencial para a multiplicação de vetores.

A Figura 4.1 mostra os tempos de execução da aplicação sequencial utilizando

compartilhando memória com dois e quatro nós, comparado ao tempo de execução de uma versão *out-of-core* da multiplicação de vetores, ou seja, lendo e gravando dados no disco. Os acessos ao disco são controlados pelo sistema operacional, e todos os dados são trazidos do disco para a memória de uma só vez pois existe memória disponível. Como pode-se observar, houve um aumento nos tempos de execução, de 243,8% para dois nós e 335,9% para quatro nós. O aumento no tempo de execução de dois para quatro nós ocorre pois o tamanho da memória global não muda. Quanto maior o número de nós, mais fragmentada fica a memória, e consequentemente menor é o tamanho da memória local. Isso implica que quanto maior o número de nós, mais falhas de página são geradas, o que explica a diferença de desempenho. O tempo gasto em falhas de página também é demasiadamante alto, ocupando mais de 93% do tempo total de execução nos dois casos. Faz-se necessária uma melhor análise desse mecanismo para entender o motivo desse tempo ser tão alto.

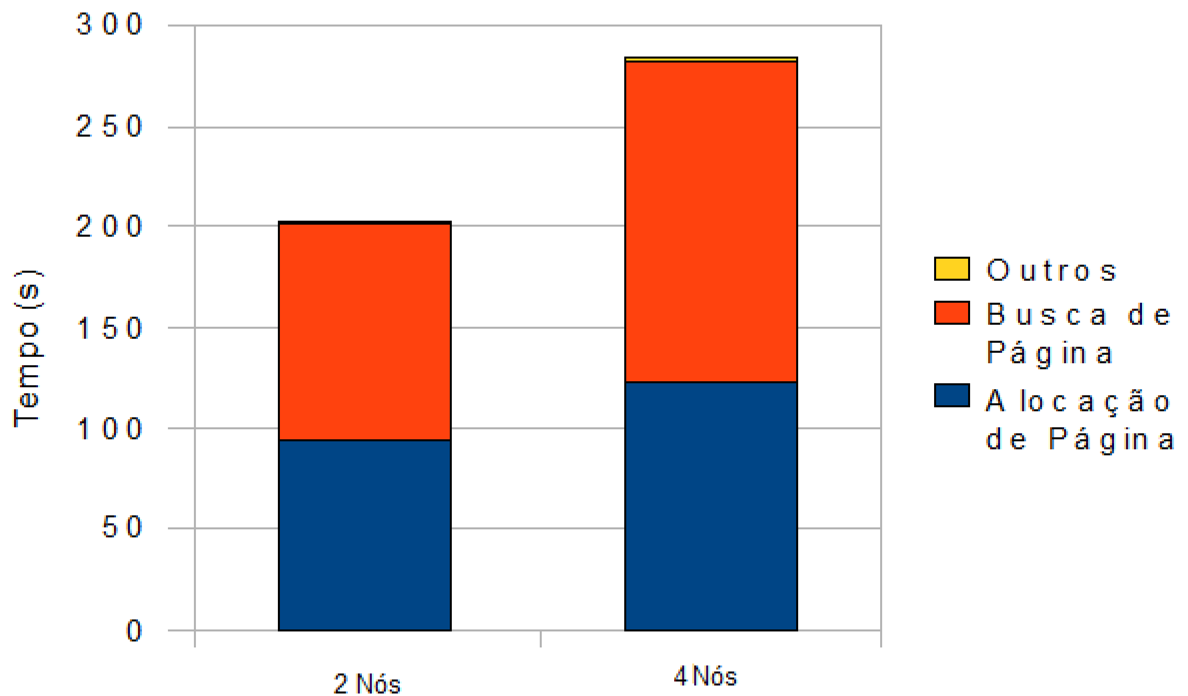

Figura 4.2: Tempo gasto em falhas de página para a multiplicação de vetores sequencial.

A figura 4.2 detalha o tempo de execução da rotina de tratamento de falhas de

página: 99% desse tempo total é gasto com alocação de páginas e com a requisição e o recebimento de páginas de outros nós. A rotina de alocação de páginas sempre verifica o número de páginas livres disponíveis no *pool*, e se o número for baixo, chama a função `kgm_oom()` para liberar memória, enviando as páginas de volta para a sua residência. Como todas as páginas, no caso de aplicações sequencias, são tratadas como escrita, todas elas precisam ser enviadas mesmo que não tenham sido modificadas. Um número grande de páginas tem que ser enviado sempre que essa função é chamada, o que faz com que ela ocupe 98% do tempo médio de alocação.

Tempos Médios (μs)	2 Nós	4 Nós
Falha de Página	833	775
Busca de Página	438	431
Alocação	391	340

Tabela 4.2: Tempos médios para a multiplicação de vetores sequencial

A razão dos tempos médios de alocação - e consequentemente o tempo de falha de página - variarem de acordo com o número de nós é que esse tempo depende diretamente do tempo gasto com a função `kgm_oom()`. Quanto mais vezes ela for invocada, mais mensagens de dados serão enviadas, influenciando diretamente o valor do tempo de alocação. Nota-se que o tempo médio gasto para buscar uma página em um nó remoto permanece praticamente constante. Por se tratar da operação mais realizada no sistema, influencia diretamente o desempenho global da aplicação.

	2 Nós	4 Nós
Número de Falhas de Página	243951	365928
Número de Mensagens	1146573	1610100

Tabela 4.3: Número de Falhas e Mensagens para a multiplicação de vetores sequencial

Como pode-se verificar na Tabela 4.3, o aumento de mais de 40% no número de falhas de página e de mensagens enviadas ocorrida ao executar a aplicação em quatro nós, ajuda a entender a queda de desempenho percebida na Figura 4.1 mostrada anteriormente.

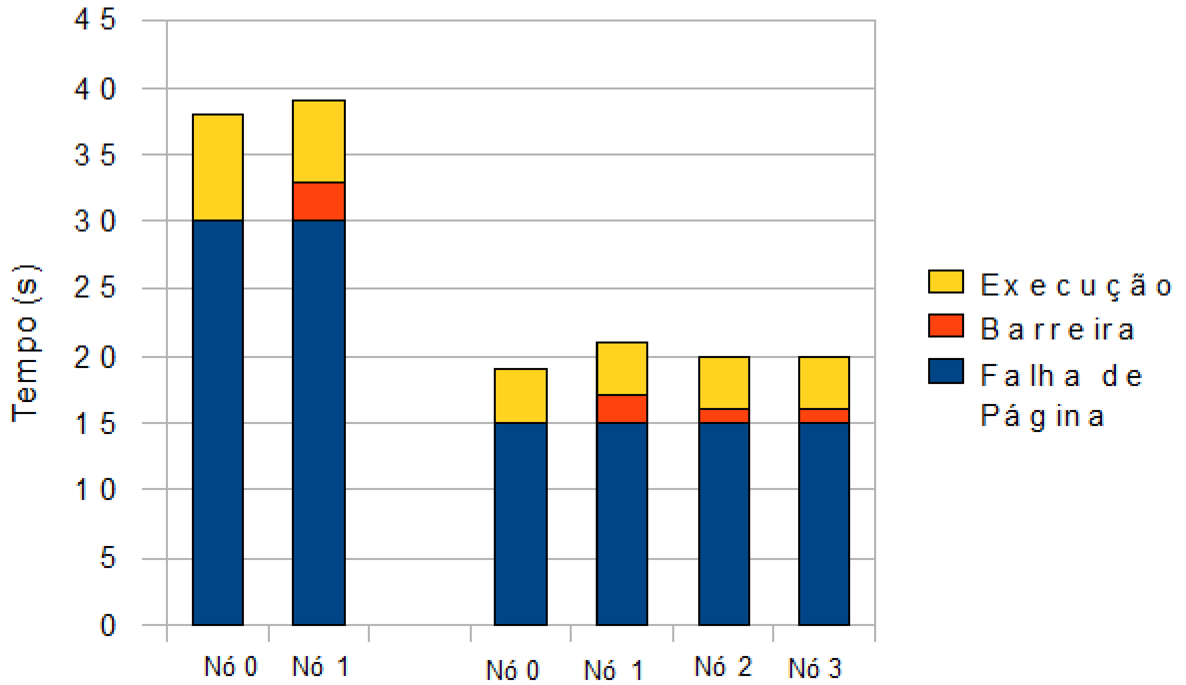

Figura 4.3: Tempo de execução paralela para a multiplicação de vetores.

Na versão paralela da multipicação de vetores, cada nó é reponsável por multiplicar um pedaço de tamanho igual do vetor. Os tempos de execução foram bem menores, como mostra a figura 4.3.

Essa redução nos tempos de execução ocorre pois, além de executar em paralelo, cada nó trabalha com um pedaço menor de memória. Isso acaba gerando um número menor de falhas de página, principalmente pelo fato da a função `kgm_oom()` não ter sido invocada nenhuma vez para as execuções realizadas, gerando menos mensagens e melhorando o desempenho da aplicação. Como a carga de trabalho é bem distribuída entre os nós, vê-se que os tempos de execução e falha de página são bem parecidos. Também por realizar a mesma quantidade de trabalho, o tempo de barreira é bem pequeno. Como a computação é realizada sobre o mesmo número de elementos de um vetor e em nós computacionais homogêneos, os tempos de execução são praticamente os mesmos.

Como é possivel notar na figura 4.4, em comparação com a versão sequencial, o tempo gasto com falhas de página diminuiu drasticamente - a função `kgm_oom` não foi chamada nenhuma vez, evitando o envio de páginas de volta para sua residência.

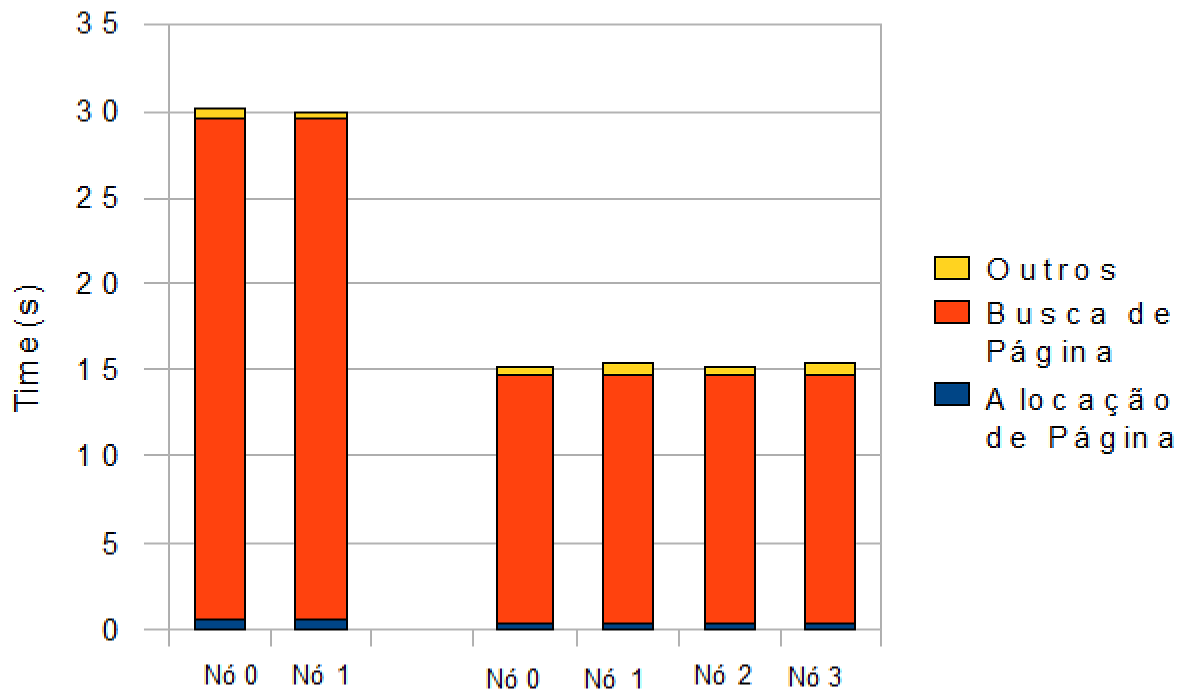

Figura 4.4: Tempo gasto em falhas de página para a multiplicação de vetores paralela.

Tempos Médios (μs)	2 Nós	4 Nós
Falha de Página	491	483
Busca de Página	476	473
Alocação	10	8

Tabela 4.4: Tempos médios para a multiplicação de vetores paralela

O tempo gasto com a busca de páginas, porém, continua sendo predominante. Os tempos médios para essa aplicação podem ser observados na tabela 4.4.

Ao dividir a computação entre mais nós, é eliminado o gargalo de processamento presente na aplicação sequencial, reduzindo em mais de 50% o número total de falhas de página no sistema e enviando quase 5 vezes menos mensagens.

	2 Nós	4 Nós
Número de Falhas de Página	121976	121976
Número de Mensagens	243954	243964

Tabela 4.5: Número de Falhas e Mensagens para a multiplicação de vetores paralela

Pela Tabela 4.5, o número de mensagens e falhas permanece constante, sendo que o número de mensagens enviadas é apenas o suficiente para satisfazer as requisições geradas pelas falhas de página. A diferença no número de mensagens enviadas, apesar do número de falhas ser igual, acontece por causa das mensagens `KGM_EXIT` enviadas no encerramento da aplicação.

4.4.2 Mergesort

A segunda aplicação escolhida para testar o protótipo foi *mergesort*. *Mergesort* é um algoritmo de ordenação, cujo conceito é combinar dois vetores já ordenados. Isto é feito dividindo recursivamente o vetor original em duas metades, até chegar a vetores com apenas 1 elemento. Nesse ponto, a recursão retorna combinando os resultados. Esta aplicação não possui um padrão de acesso bem definido como a aplicação de multiplicação de vetores apresentada, e oferecerá um conjunto diferente de resultados para serem analisados.

Resultados

Assim como na aplicação apresentada anteriormente, são analisadas as versões sequencial e paralela dessa aplicação. O vetor a ser ordenado possui um tamanho total de 952Mb de memória (243.952 páginas).

Segundo a figura 4.5, em comparação com a versão *out-of-core* da aplicação (mais uma vez os acessos ao disco são controlados pelo sistema operacional, com todos os dados carregados para a memória de uma só vez), pode-se notar um aumento de

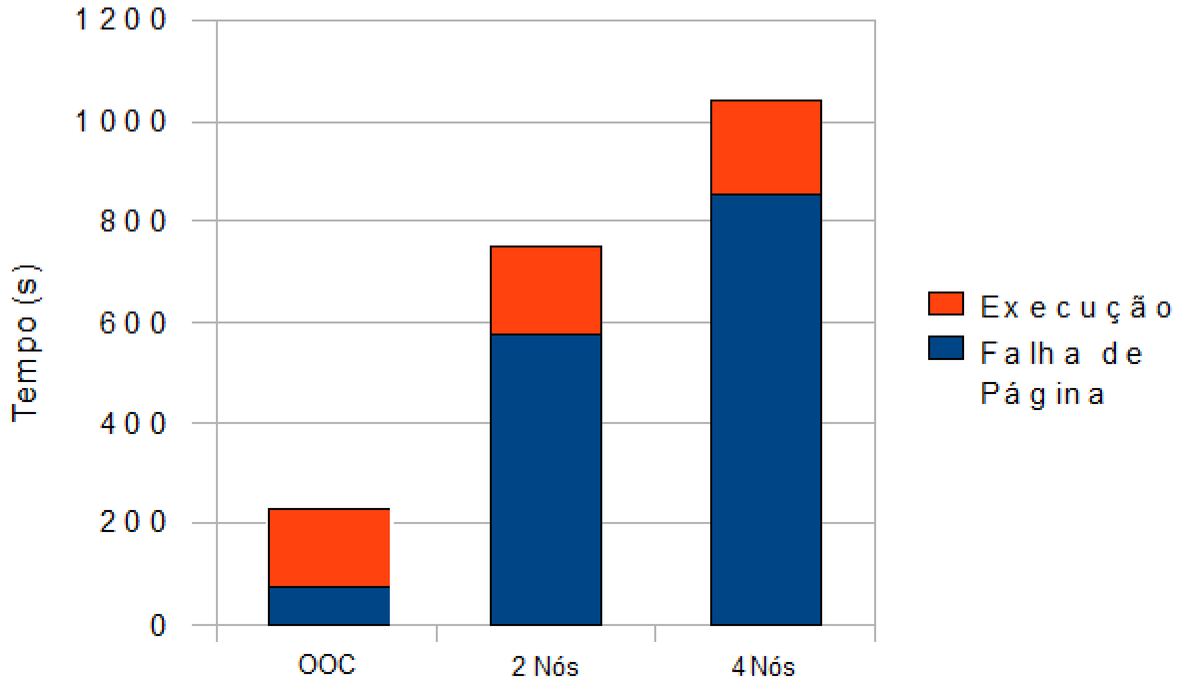

Figura 4.5: Tempo de execução sequencial para mergesort.

350% no tempo de execução sequencial com dois nós e 484% para quatro nós. Vê-se novamente que o tempo gasto com a rotina de tratamento de falhas de página é alto, consumindo mais de 76% do tempo total de execução.

Assim como na versão sequencial da multiplicação de vetores, o tempo médio de falha da página é bastante alto. Como mostra a Tabela 4.6, o tempo médio de busca de página, que tem se mostrado constante em todos os experimentos e tem sido a maior fatia de tempo na execução da falha de página, aqui é alcançado pelo tempo de alocação. De acordo com os resultados analisados anteriormente, pode-se concluir que o aumento no tempo de alocação se deve a um número maior de chamadas à função `kgm_oom()`.

Tempos Médios (μs)	2 Nós	4 Nós
Falha de Página	870	857
Busca de Página	433	440
Alocação	434	414

Tabela 4.6: Tempos médios para mergesort sequencial

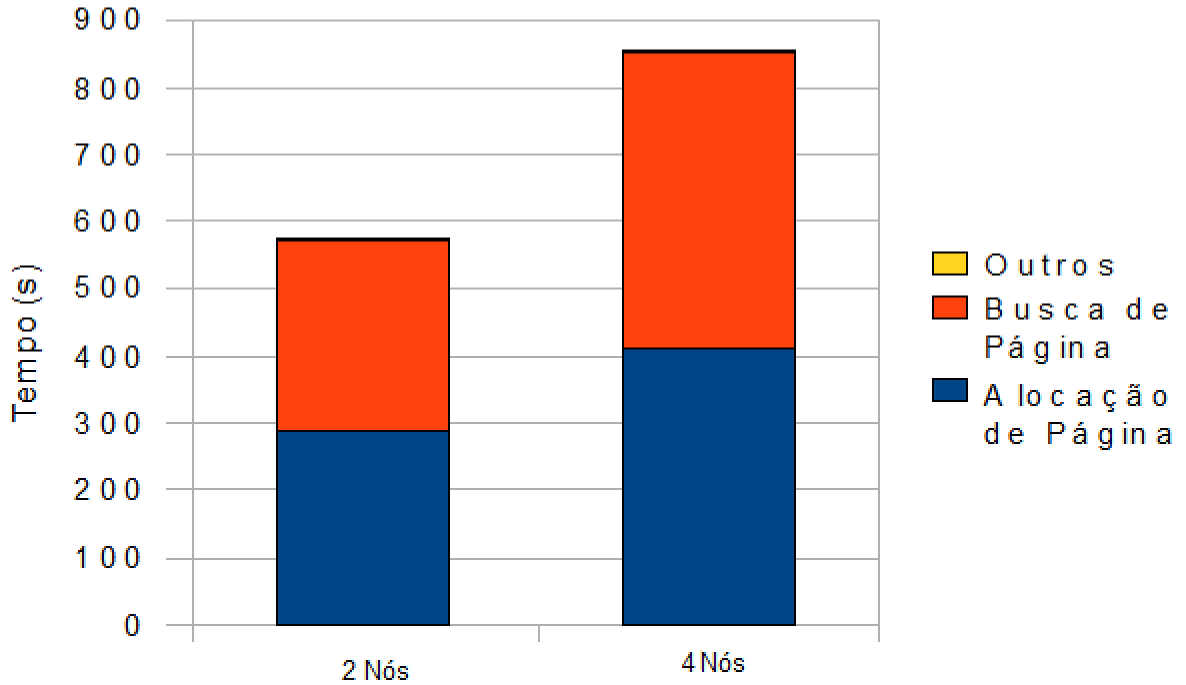

Figura 4.6: Tempo gasto em falhas de página para mergesort sequencial.

De fato, no mergesort sequencial, a função `kgm_oom()` foi chamada três vezes mais do que na aplicação de multiplicação de vetores, resultando em um número cerca de três vezes mais alto de mensagens enviadas.

	2 Nós	4 Nós
Número de Falhas de Página	659619	1000193
Número de Mensagens	3295249	4854649

Tabela 4.7: Número de Falhas e Mensagens para mergesort sequencial

Na versão paralela do mergesort, a execução da aplicação é dividida em rodadas. Na primeira, todos os nós ordenam uma parte do vetor correspondente aos seus dados locais, de modo que não há necessidade de buscar dados em outros nós. Com essas partes ordenadas, nas rodadas seguintes será feito o merge dos vetores. Para dois nós, o nó zero realizará esse trabalho. Para quatro nós, os nós 0 e 2 fazem o merge dos vetores contidos nos nós 0 e 1 e 2 e 3 respectivamente, para então o nó 0 realizar o último merge.

Para que esse merge funcione, é necessário que os merges realizados anteriormente

pelos nós 0 e 2 sejam realizados utilizando lock, para que os dados ordenados sejam atualizados em suas residências. É importante notar que a execução dessa aplicação não é balanceada.

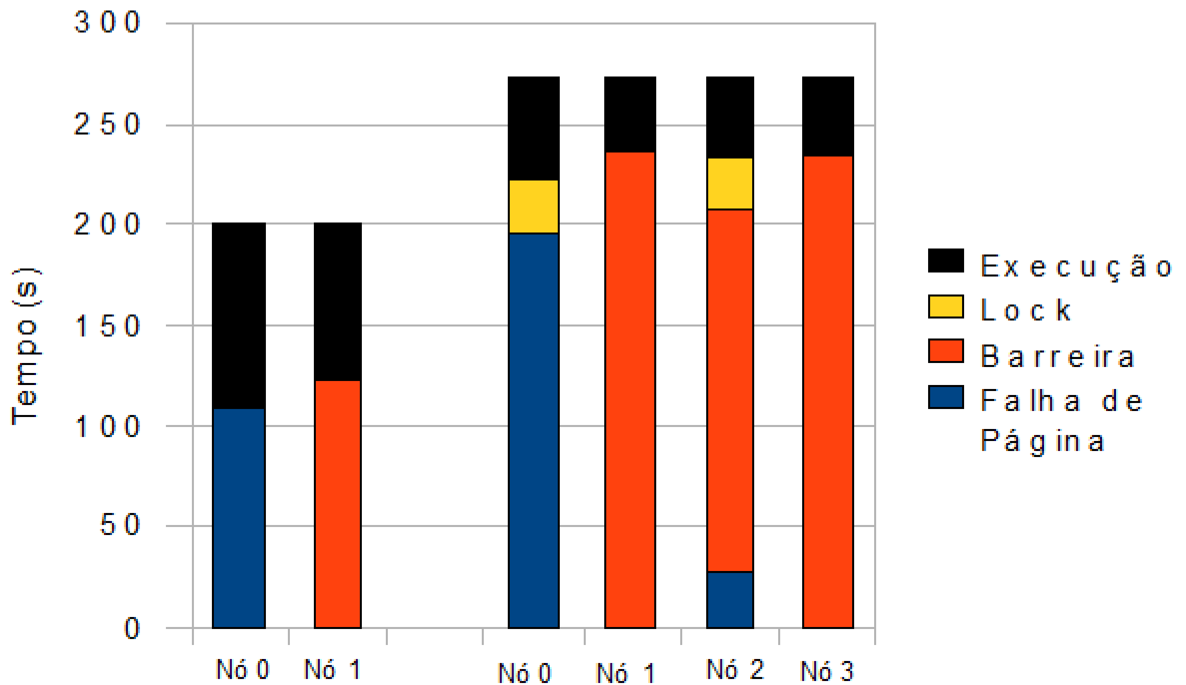

Figura 4.7: Tempo de execução paralela para mergesort.

Na figura 4.7, pode-se perceber o tempo de lock ao executar a aplicação com quatro nós. Apenas um lock é usado na aplicação, e seu tempo de execução elevado se deve ao fato de ele enviar todas as páginas acessadas entre as funções de `lock_acquire()` e `lock_release()` de volta para suas residências, mantendo-as atualizadas. Esse processo de atualização das páginas ocupa 99% do tempo total de lock. Como os nós realizam cargas diferentes de trabalho para ordenar o vetor, os que realizam menos trabalho acabam esperando mais tempo nas barreiras.

A Figura 4.8 ilustra o comportamento das falhas de página para essa execução. Os nós 1 e 3, que só executam a primeira rodada da aplicação, não gastam tempo praticamente nenhum com elas. Por outro lado, os nós 0 e 2, a partir da segunda rodada, passam a trabalhar com dados presentes em outros nós.

O tempo total gasto com falhas de página resume-se à praticamente o seu tempo de busca. Ao contrário do tempo de alocação, que varia bastante pois depende da

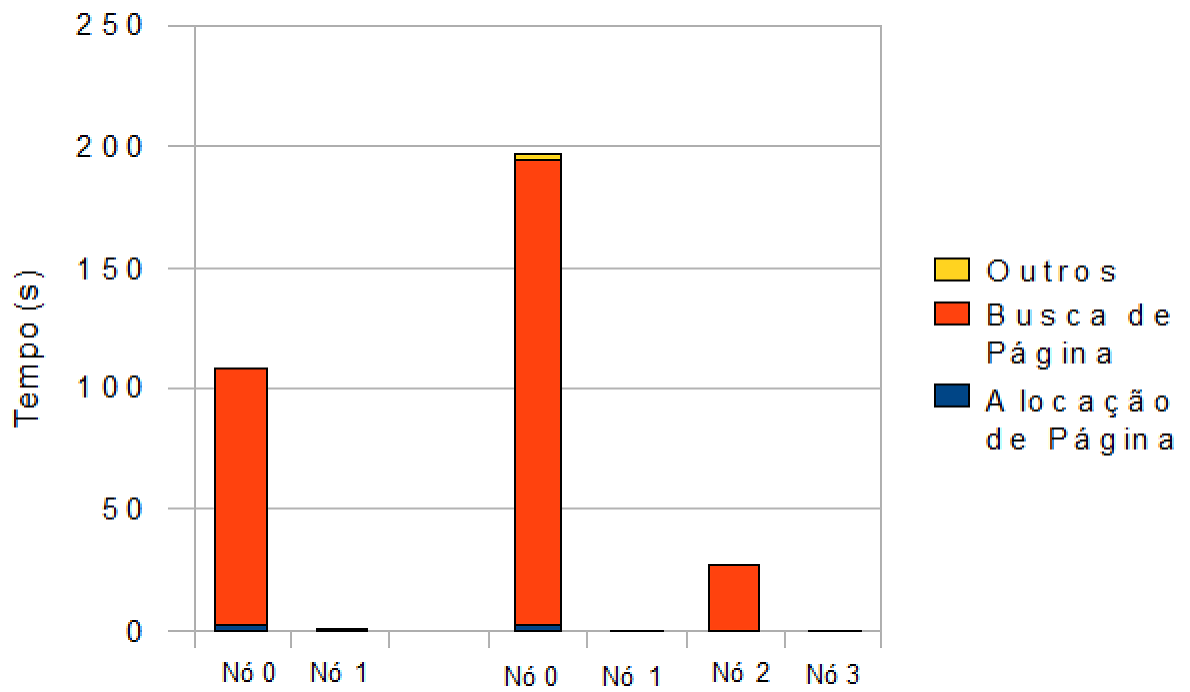

Figura 4.8: Tempo gasto em falhas de página para mergesort paralelo.

Tempos Médios (μs)	2 Nós	4 Nós
Falha de Página	446	458
Busca de Página	435	443
Alocação	7	6

Tabela 4.8: Tempos médios para mergesort paralelo

disponibilidade de páginas do *pool* e da função de liberação de páginas, o tempo médio de busca se manteve constante durante todos os testes realizados.

	2 Nós	4 Nós
Número de Falhas de Página	243952	487908
Número de Mensagens	487904	1341234

Tabela 4.9: Número de Falhas e Mensagens para mergesort paralelo

De acordo com a Tabela 4.9, pode-se novamente perceber a drástica redução no número total de mensagens enviadas pela versão paralela, ainda que executando de forma assimétrica, em comparação com a sequencial.

4.5 Análise

A latência para buscar uma página é uma métrica importante. A média de 440 μs pode ser melhorada, pois combinada com o número alto de mensagens, acaba por deteriorar o desempenho do sistema. A Tabela 4.10 divide o tempo médio de busca em *tempo de envio*, *tempo de transmissão* e *tempo de recepção* para as mensagens de `PAGE_REQUEST` e de `PAGE_REPLY`. É mostrado ainda o *tempo de processamento*, que corresponde ao tempo gasto para processar o pedido da página, ou seja, o tempo gasto entre a recepção de um `PAGE_REQUEST` e o envio de um `PAGE_REPLY`.

	Tempo de Envio (μs)	Tempo de Transmissão (μs)	Tempo de Recepção (μs)	Tempo Total (μs)
`PAGE_REQUEST`	8	83	53	144
`PAGE_REPLY`	18	142	127	287
Tempo de Processamento	-	-	-	9

Tabela 4.10: Tempo de busca.

A diminuição do tempo de busca, contudo, apenas atenuaria o problema. Através dos testes realizados, pode-se perceber claramente que o grande responsável pelo mau desempenho do sistema é o alto número de mensagens enviadas, e este é o ponto principal a ser focado.

O alto número de mensagens se deve ao fato do sistema não utilizar falha de proteção. O tamanho do *pool* utilizado também influencia diretamente nesse fator, pois quanto menor o *pool*, mais rápido ele ficará sem páginas livres e precisará ser esvaziada, através do envio de páginas para seus nós home. Ao usar falha de proteção, o sistema teria como saber se uma determinada página está sendo acessada para leitura ou para escrita. Assim, durante o esvaziamento do *pool* através da função `kgm_oom()`, as páginas acessadas para escrita não precisariam ser enviadas, diminuindo bastante o número total de mensagens.

Tomando a aplicação sequencial de multiplicação de vetores com dois nós como exemplo: o nó 0, que realiza a computação, é o home do primeiro vetor, que vai receber o resultado da multiplicação e, portanto, ser escrito. Todos os acessos ao outro vetor, no nó 1, são de leitura. Como a função de liberação de páginas foi chamada três vezes, e o número de páginas livres no *pool* é de 73186, o número de mensagens enviadas desnecessariamente é de 658674, já que são necessárias três mensagens por cada página atualizada(uma de conrole, uma de dados e uma de *ack*). Isso equivale a 57,4% das mensagens desse cenário, o que identifica claramente o principal problema do sistema.

Capítulo 5

Trabalhos Relacionados

Diversos trabalhos têm proposto mecanismos para explorar o uso da memória em clusters de computadores. Dois tipos de aplicações comuns são o uso das memórias remotas para *swap* e sistemas de memória compartilhada distribuída em software. Trabalhos relacionados a estes dois métodos serão descritos a seguir.

5.1 Swap Sobre a Rede

Global Memory Service [20] implementa um esquema de swap sobre a rede, gerenciando a memória dos nós cluster como um todo. Quando um nó fica sem memória, a decisão sobre qual página deve ser substituída é feita através de informações globais, isto é, baseia-se na memória disponível em todos os nós do cluster para a substituição de páginas localmente. Em cada nó é alocada uma área de memória global, que pode crescer ou diminuir de acordo com a carga do nó. Essa área não possui tamanho fixo localmente, mas a soma das áreas em todos os nós tem sempre o mesmo tamanho. GMS não possui informações sobre o programa sendo executado. Isto quer dizer que a gerência de memória é feita de modo a otimizar a distribuição de memória no cluster, e não o desempenho da aplicação.

5.2 Memória Compartilhada Distribuída em Software

JiaJia [8] é um sistema DSM capaz de agregar as memórias de vários nós e formar um espaço de endereçamento compartilhado maior do que o tamanho da memória local. Usa a consistência de escopo [22], implementada através de um protocolo de coerência de cache baseado em locks. Assim como a maioria dos sistemas DSMs, é implementado como uma biblioteca e executado em nível de usuário.

MOMEMTO (*MOre MEMory Than Others*)[24] é também um sistema DSM capaz de agregar as memórias de vários nós, acessando-as através de um espaço de endereçamento compartilhado. É implementado no kernel 2.4 do Linux, através de módulos. MOMEMTO não implementa nenhum protocolo de coerência, mas oferece uma função onde o programador pode explicitamente atualizar uma página, enviando-a para sua residência, que repassará os dados atualizados aos nós que tiverem uma cópia da página.

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Conclusões

Esta dissertação propôs, implementou e avaliou um sistema em modo kernel capaz de oferecer um espaço de endereçamento compartilhado para agregar as memórias de um *cluster*. Devido a problemas na implementação, o objetivo de agregar memórias não pode ser atingido, pois o maior espaço de endereçamento alocado não passava de 1Gb. Apesar desse problema, o protótipo criado foi avaliado e os resultados não mostraram desempenho satisfatório. A latência de buscar uma página em um nó remoto pode ser diminuída. Com uma latência mais baixa o desempenho do sistema seria diretamente beneficiado, principalmente para a execução de aplicações *out-of-core*, já que a execução sequencial dessas aplicações mostrou que o nó que realiza computação é um gargalo. Esgotando as páginas do *pool* e liberando-as inúmeras vezes, o número de mensagens é bastante elevado e depende de baixas latências na troca de mensagens para ser eficaz. O principal problema, contudo, foi usar apenas falhas de página no sistema. Por não usar falhas de proteção, não há como saber se uma página foi acessada para escrita ou leitura. Assim páginas que somente foram lidas acabam por ser enviadas de volta para suas residências, mesmo sem terem sido alteradas, o que gera um número muito alto de mansagens na rede. Conforme discutido no Capítulo 4, um exemplo tomado identificou uma porcentagem de quase 60% de mensagens enviadas desnecessariamente. Apenas a diminuição desse número de mensagens não é suficiente para que a aplicação execute mais rápido do que a *out of core*. Porém, acredita-se que, com a diminuição da latência das mensagens, o

sistema conseguiria ser eficiente mesmo para este tipo de aplicação.

6.2 Trabalhos Futuros

Um trabalho que merece ser pesquisado futuramente é o uso de falha de proteção juntamente com as falhas de páginas, já utilizadas no modelo atual. Isso melhoraria consideravelmente o controle sobre os acessos às páginas. Na prática, as funções que implementam o lock seriam diretamente beneficiadas, executando mais rápido e gerando menos falhas de página. A função `kgm_oom` também seria beneficiada ao enviar para as residências somente as páginas que foram alteradas, reduzindo assim o número de mensagens enviadas.

Um estudo interessante a ser feito seria uma implementação em uma versão de 64 bits do kernel, que oferece um espaço de endereçamento maior do que do kernel de 32 bits com PAE.

Para reduzir a latência de envio e recepção de mensagens, um estudo pode ser feito para otimizar os sockets de acordo com os tamanhos das mensagens enviadas.

Referências Bibliográficas

[1] "TOP500 Supercomputer Sites", `http://www.top500.org/`, Acessed on July 2010.

[2] WEISS, A., "Computing in the clouds", *netWorker*, v. 11, n. 4, pp. 16–25, 2007.

[3] ROURE, D. D., BAKER, M. A., JENNINGS, N. R., "The evolution of the grid". In: *Grid Computing: Making the Global Infrastructure a Reality*, pp. 65–100, John Wiley & Sons, 2003.

[4] STERLING, T., BECKER, D. J., SAVARESE, D., et al., "Beowulf: A Parallel Workstation For Scientific Computation". In: *In Proceedings of the 24th International Conference on Parallel Processing*, pp. 11–14, CRC Press, 1995.

[5] BAL, H. E., KAASHOEK, M. F., TANENBAUM, A. S., "Orca: A language for parallel programming of distributed systems", *IEEE Transactions on Software Engineering*, v. 18, pp. 190–205, 1992.

[6] VELDEMA, R., BHOEDJANG, R., BAL, H., "Jackal, A Compiler Based Implementation of Java for Clusters of Workstations". In: *in Proc. of PPoPP*, 2001.

[7] KELEHER, P., COX, A. L., DWARKADAS, S., et al., "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems". In: *IN PROCEEDINGS OF THE 1994 WINTER USENIX CONFERENCE*, pp. 115–131, 1994.

[8] HU, W., SHI, W., TANG, Z., "JIAJIA: A Software DSM System Based on a New Cache Coherence Protocol". In: *In Proceedings of the High-Performance Computing and Networking Conference*, pp. 463–472, 1999.

[9] VITTER, J. S., "External memory algorithms and data structures: dealing with massive data", *ACM Comput. Surv.*, v. 33, n. 2, pp. 209–271, 2001.

[10] LOVE, R., *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press, 2005.

[11] TORRELLAS, J., LAM, M. S., HENNESSY, J. L., "False sharing and spatial locality in multiprocessor caches", *IEEE Transactions on Computers*, v. 43, pp. 651–663, 1994.

[12] LI, K., HUDAK, P., "Memory Coherence in Shared Virtual Memory Systems", 1989.

[13] KELEHER, P., COX, A. L., ZWAENEPOEL, W., "Lazy Release Consistency for Software Distributed Shared Memory". In: *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 13–21, 1992.

[14] BOVET, D., CESATI, M., *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[15] GORMAN, M., *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR: Upper Saddle River, NJ, USA, 2004.

[16] "The GNU General Public License", `http://www.gnu.org/licenses/gpl.html`, Acessed on July 2010.

[17] KRAMER, W. T. C., CRAWER, J. M., "Effective use of Cray supercomputers". In: *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pp. 721–731, ACM: New York, NY, USA, 1989.

[18] "Message Passing Interface (MPI) Forum Home Page", `http://www.mpi-forum.org/`, Acessed on August 2010.

[19] "PVM: Parallel Virtual Machine", `http://www.csm.ornl.gov/pvm/`, Acessed on August 2010.

[20] FEELEY, M. J., MORGAN, W. E., PIGHIN, E. P., et al., "Implementing global memory management in a workstation cluster". In: *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp. 201–212, ACM: New York, NY, USA, 1995.

[21] ZHOU, Y., IFTODE, L., LI, K., "Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems", *SIGOPS Oper. Syst. Rev.*, v. 30, n. SI, pp. 75–88, 1996.

[22] IFTODE, L., SINGH, J. P., LI, K., "Scope Consistency : A Bridge between Release Consistency and Entry Consistency". In: *In Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 277–287, 1996.

[23] AMZA, C., COX, A. L., ZWAENEPOEL, W., et al., "Software DSM Protocols that Adapt between Single Writer and Multiple Writer". In: *HPCA '97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, p. 261, IEEE Computer Society: Washington, DC, USA, 1997.

[24] TREVISAN, T., COSTA, V., WHATELY, L., et al., "Distributed Shared Memory in Kernel Mode". In: *SBAC-PAD '02: Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing*, p. 159, IEEE Computer Society: Washington, DC, USA, 2002.

[25] "The Linux Kernel Archives", `http://www.kernel.org/`, Acessed on September 2010.