UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MÁRCIA CARGNIN MARTINS GIRALDI

MMG: Um Ambiente de Memória Compartilhada Distribuída para uma Arquitetura Cluster Linux

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Dr. Luís Fernando Friedrich

Florianópolis, Agosto de 2000.

MMG: Um Ambiente de Memória Compartilhada Distribuída para uma Arquitetura Cluster Linux

MÁRCIA CARGNIN MARTINS GIRALDI

Esta Dissertação foi julgada adequada para obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Banca Examinadora

Luis Fernando Friedrich, Dr

Fernando A. Ostuni Gauthier,Dr

Thadeu Botteri Corso, Dr

José Mazzuco Jr, Dr

Carlos Barros Montez, Dr

À Laura, Alexandre e Marcos

Agradeço

A Deus pela vida;

Aos meus pais, que me ensinaram a viver e a ser o que sou;

Ao meu esposo e filhos, que me mostraram o significado da vida.

À ACAFE, UNISUL e UFSC, instituições cujo apoio foram fundamentais para o desenvolvimento deste trabalho.

Aos professores do curso de Pós-Graduação em Ciência da Computação e, especial agradecimento ao Prof. Dr. Luis Fernando Friedrich, pela dedicação na orientação e condução deste trabalho aos seus objetivos.

Ao Professor Dr. Carlos Barros Montez , que com empenho e dedicação, contribuiu para que o objetivo do trabalho fosse alcançado com êxito.

Ao acadêmico Fernando José Cardoso, por sua dedicação e entusiasmo na codificação do protótipo.

SUMÁRIO

RESUMO

A grande evolução tecnológica têm permitido uma melhora na velocidade de processamento, mas sempre dentro de limites físicos bem estabelecidos. As redes de alta velocidade permitem a conexão de dezenas ou até centenas de máquinas, com altas taxas de transferência. O resultado da aplicação desta tecnologia é o fato de hoje ser simples construir sistemas de computação compostos por um grande número de processadores interligados através de redes de alta velocidade. Portanto, a exploração do processamento paralelo e distribuído é uma das formas de ampliar os limites de desempenho dos sistemas computacionais.

Um sistema de Memória Compartilhada Distribuída (MCD) oferece a abstração de um espaço de endereçamento lógico, a que todos os processadores de uma arquitetura distribuída têm acesso, apesar de fisicamente a memória ser local a cada um deles. Os sistemas MCD controlam a distribuição física dos dados, oferecendo ao programador acesso transparente a memória virtual compartilhada.

A característica dos modelos MCD de fornecer uma interface transparente e um ambiente de programação conveniente para aplicações paralelas e distribuídas tem

feito o assunto foco de numerosas pesquisas nos últimos anos.

Este trabalho apresenta o projeto de um ambiente MCD para um *cluster* com sistema operacional Linux. O sistema foi baseado nas primitivas de memória compartilhada da API (interface de programação de aplicações) do UNIX referentes a comunicação entre processos (IPC – *Interprocess Comunication*). As primitivas originais sofreram modificações para suportar coerência e sincronização distribuídas. Dessa forma, todo o ambiente de MCD fica executando a nível do sistema operacional.

ABSTRACT

The great technological evolution has allowed an improvement of the speed processing, but always inside of well established physical limits. The networks of high speed allow the connection of sets of ten or until hundreds of machines, with high rates of transference. The result of the application of this technology is the fact that today is simple to construct computer systems composed of great number of linked processors through networks of high speed. Therefore, the scanning of the parallel and distributed processing is one of the forms to extend the limits of the computational systems performance.

A system of Memória Compartilhada Distribuída (MCD) offers the abstraction of a logical addressing space, the one that all the processors of a distributed architecture have access, although physically the memory is local to each one of them. MCD Systems control the physical distribution of the data, offering to the programmer transparent access to the shared virtual memory.

The feature of MCD models in order to supply a transparent interface and an environment of convenient programming applications distributed parallel bars and has made the subject focus of numerous research in the last years.

This work presents the design of an environment MCD to cluster with operational system Linux. The system was based on the primitive of shared memory of UNIX API (interface of programming of applications) referring the communication between processes (IPC - Interprocess Comunication). The original primitive had suffered modifications to support distributed coherence and synchronization. Thus, all the MCD environment is executed in the operational system level.

LISTA DE FIGURAS

LISTA DE TABELAS

INTRODUÇÃO

O maior objetivo do processamento paralelo e distribuído é aumentar o desempenho de aplicações que necessitam de grande poder computacional e, que são pouco eficientes quando executadas seqüencialmente. Para tanto, o aumento de desempenho é obtido particionando-se um processo em processos menores, alocando-os simultaneamente em diferentes processadores.

Sistemas com vários processadores podem ser classificados em função do grau de integração e acoplamento dos seus componentes: os *multiprocessadores* possuem espaço de endereçamento compartilhado por um pequeno número de processadores; nos *multicomputadores*, cada processador possui sua própria memória privativa e canais de comunicação ligados através de redes compostas por múltiplos canais bipontuais; *redes de computadores* são formadas por computadores independentes interconectados através de redes entendidas como canais de comunicação compartilhados.

A comunicação entre processos pode ser feita de duas formas: *memória compartilhada* ou *passagem de mensagem*. O paradigma de comunicação por passagem

de mensagem é mais natural de ser usado em multicomputadores e redes de computadores, por não possuírem memória fisicamente compartilhada. Os multiprocessadores podem utilizar naturalmente comunicação por memória compartilhada.

Os multiprocessadores são mais difíceis para serem construídos e, requerem uma tecnologia de *hardware* de maior custo. No entanto, são mais fáceis para programar. Os multicomputadores, ao contrário, possuem um custo de fabricação mais baixo e são mais fáceis de serem projetados, porém apresentam uma programação bem mais difícil. A partir desta realidade, surgiu que conceito de *Memória Compartilhada Distribuída* com o objetivo de fornecer um ambiente de fácil programação para os sistemas que não possuem memória fisicamente compartilhada.

Memória Compartilhada Distribuída (MCD) é uma abstração utilizada para compartilhar dados entre processos que não possuem memória fisicamente compartilhada.

1.1 - Objetivos

O presente trabalho individual tem como objetivo realizar estudo sobre Memória Compartilhada Distribuída, conceitos e aplicações, a fim de propor o estudo de algumas alternativas para se criar um ambiente MCD, para uma arquitetura *Cluster Linux*.

O uso de *clusters* com Linux tem crescido devido a ser uma alternativa de baixo custo para processamento paralelo e sistema operacional gratuito de código aberto.

CAPÍTULO II

CONCEITOS RELACIONADOS À MCD

2.1 - Arquitetura e Motivação

Memória Compartilhada Distribuída (MCD) é uma abstração usada para compartilhar dados entre computadores que não compartilham memória física. Processos acessam MCD por operações de leituras e alterações, através do que parece ser uma memória comum, dentro dos seus espaços de endereçamento. Em tempo de execução, o sistema assegura a transparência desses acessos, onde processos, mesmo executando em computadores diferentes, observam as atualizações feitas por outros e vice-versa. Do ponto de vista dos processos é como se estes tivessem acesso a uma única memória compartilhada, mas de fato a memória física é distribuída (Figura 2.1).

A grande vantagem do uso de MCD é que o programador se abstrai das preocupações com passagem de mensagem. A programação distribuída por passagem de mensagem dificulta o trabalho do programador, pois este tem que se ater a detalhes de comunicação, além de tornar o programa mais complexo [TAN95]. MCD é, principalmente, uma abstração para aplicações paralelas ou para qualquer aplicação ou

1.2 - Divisão do Trabalho

Este trabalho conta com mais quatro capítulos, além deste capítulo introdutório. O Capítulo 2 apresenta os conceitos básicos de MCD, além de outros conceitos importantes para o entendimento dos demais capítulos. O Capítulo 3 faz um estudo de soluções envolvendo MCD existentes, trazendo à tona as principais características, vantagens e desvantagens de cada sistema descrito. Finalmente, no Capítulo 4, encontra-se a proposta de continuidade da pesquisa visando o trabalho de Dissertação de Mestrado.

grupo de aplicações distribuídas em que um item individual de dado compartilhado pode ser acessado diretamente.

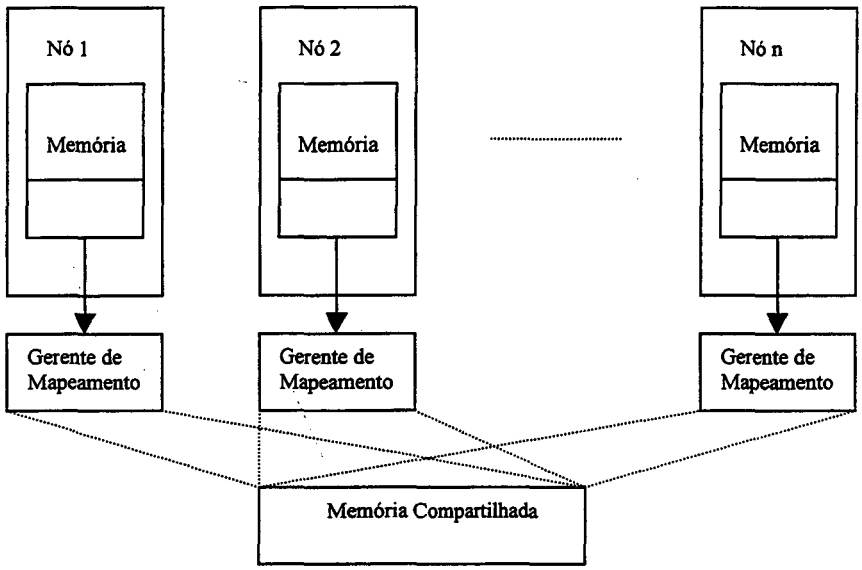

Figura 2.1 – Memória Compartilhada Distribuída [LI89]

As trocas de mensagens não podem ser evitadas completamente em um sistema distribuído: na ausência de memória fisicamente compartilhada, a MCD usa a troca de mensagens para enviar atualizações entre computadores. Uma forma de melhorar o desempenho do sistema, evitando uma constante troca de mensagem, é manter em cada computador uma cópia local de itens de dados, recentemente acessados.

MCD é uma área ativa de pesquisa desde a década de 1980. Um dos

primeiros exemplos notáveis de uma implementação de MCD foi o sistema de arquivos *Apollo Domain* [COU95], no qual processos em diferentes estações de trabalho compartilham arquivos, mapeando-os simultaneamente em seus espaços de endereçamento. O significado de MCD cresceu ao lado do desenvolvimento de multiprocessadores de memória compartilhada. Muita pesquisa foi realizada para desenvolver algoritmos satisfatórios para computação paralela em multiprocessador e, no uso de *hardware* de *caching* para maximizar o número de processadores que podem ser sustentados por eles. Porém, a prática limita em aproximadamente 64 ou menos processadores, onde estes são conectados a módulos de memória sobre um barramento comum [HWA93].

Podemos salientar algumas diferenças entre os dois paradigmas de comunicação entre processos, passagem de mensagem e MCD, descritas a seguir.

2.2 - Diferenças entre os paradigmas de passagem de mensagem e MCD

Podemos relacionar algumas diferenças entre troca de mensagem e MCD como mecanismos de comunicação e sincronização (Tabela 2.2):

Modelo de programação: de acordo com o modelo de passagem de mensagem, variáveis têm que ser ordenadas por um processo, transmitidas e desempacotadas em outras variáveis no processo receptor. Já, com memória compartilhada, os processos fazem referência às variáveis compartilhadas diretamente, assim nenhuma ordenação é necessária. A maioria das implementações permite variáveis armazenadas em MCD para serem declaradas e acessadas similarmente às variáveis não compartilhadas normais. A troca de mensagem, por outro lado, permite a

comunicação entre processos, enquanto permite a proteção dos espaços de endereçamento privados, considerando que processos que utilizam MCD podem, por exemplo, causar um ao outro falhas, alterando dados erroneamente.

Sincronização entre processos é alcançado no modelo de mensagem pelas primitivas de passagem de mensagem. No caso de MCD, sincronização normalmente é implementada através de programação com bloqueios (*locks*) e semáforos, que requerem implementação diferente no ambiente distribuído.

Eficiência: experiências mostram que certos programas paralelos desenvolvidos para MCD, podem ter um desempenho superior aos programas funcionalmente equivalentes, escritos para troca de mensagem na mesma plataforma de hardware - pelo menos no caso de pequeno número de computadores. Porém, este resultado não pode ser generalizado. O desempenho de um programa baseado em MCD depende de muitos fatores que serão mostrados ainda neste capítulo.

Há uma diferença visível nos custos associados com os dois tipos de programação. Na troca de mensagem, todo o acesso a dados remoto está explícito, então o programador sempre está atento se uma operação particular é interna ao processo ou se envolve o custo de comunicação. Porém, usando MCD qualquer leitura ou atualização pode ou não envolver comunicação. Se faz ou não, depende de certos fatores como: se os dados foram acessados anteriormente, e o padrão de compartilhamento entre processos em computadores diferentes.

Não há nenhuma resposta definitiva sobre se MCD é preferível a troca de mensagem para qualquer aplicação particular. MCD é uma ferramenta promissora que em última instância depende da eficiência com que pode ser implementada.

Passagem de Mensagem	MCD
Exige ordenação de variáveis	As variáveis podem ser acessadas diretamente
Espaço de endereçamento protegido	Espaço de endereçamento aberto
Sincronização por troca de mensagem	Sincronização por *lock* e semáforos
Não persistente	Persistente
Acesso remoto explícito	Acesso remoto transparente

Tabela 2.2 - Passagem de mensagem X MCD

2.3 - Principais abordagens na implementação de MCD

Existem três abordagens principais na implementação de Memória Compartilhada Distribuída que respectivamente envolve o uso de *hardware*, memória virtual e biblioteca (Figura 2.3), que não são, necessariamente, mutuamente exclusivos:

Suporte Hardware: Algumas arquiteturas de multicomputador (por exemplo, Dash e PLUS) possuem *hardware* especializado para gerenciar carga e guardar instruções utilizadas em endereços de MCD e, possibilitar a comunicação com módulos de memória remotos.

Memória Virtual: Ivy [1989], Munin [1991], Mirage [1989], Clouds [1991], Choices [1990], COOL [1993] e Mether [1989] são sistemas que implementam MCD como uma região de memória virtual que ocupa a mesma faixa de endereço no espaço de endereçamento de todo processo participante. Em cada caso o núcleo do sistema

operacional mantém a consistência de dados dentro de regiões de MCD como parte de manipulação de falta de página.

Suporte de Biblioteca: Algumas linguagens ou extensões de linguagem como Orca [1990] e Linda [1989] são formas de apoio a MCD. Neste tipo de implementação o compartilhamento não é implementado pelo sistema de memória virtual, mas por comunicação entre instâncias com o apoio da linguagem. Os processos quando compilados, são acrescidos de módulos da biblioteca onde eles fazem acesso a itens de dados da MCD. As bibliotecas têm acesso a itens de dados locais e, fazem a comunicação necessária para manter a consistência do dado. Processos também podem executar *threads* para tratar mensagens recebidas que pedem operações relacionadas a MCD.

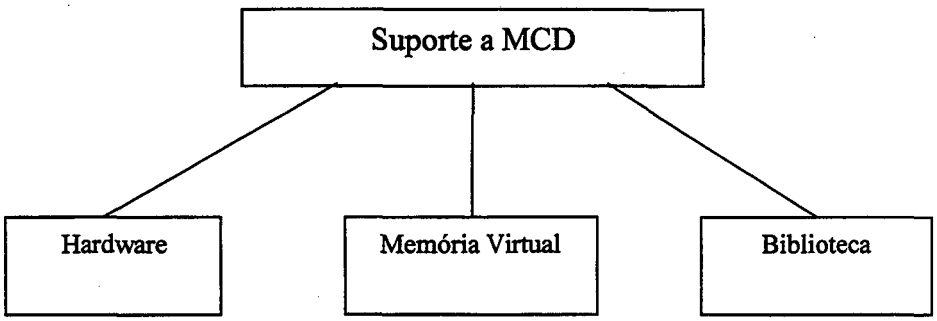

Figura 2.3 – Principais abordagens na implementação de MCD

2.4 - Algoritmos para implementação de MCD

Os algoritmos para implementar MCD tratam com dois problemas básicos: como distribuir, estática ou dinamicamente, os dados compartilhados através do sistema, para minimizar a latência de acesso; e como preservar a coerência dos dados

compartilhados, sem prejudicar o desempenho.

Duas estratégias utilizadas para distribuição dos dados compartilhados são *replicação* e *migração*. Replicação permite que várias cópias do mesmo item de dado residam em diferentes memórias locais (ou *caches*). É utilizada principalmente para habilitar acessos simultâneos de diferentes nodos ao mesmo dado, predominantemente quando o compartilhamento para leitura prevalece. Migração implica que somente uma cópia de um item de dado existe no sistema, portanto os itens de dados são movidos sob demanda para uso exclusivo. Com o objetivo de diminuir o *overhead* de gerenciamento de coerência, usuários preferem esta estratégia quando padrões seqüenciais de compartilhamentos para escrita prevalecem.

Para permitir acesso simultâneo, MCD faz uso de replicação de dados, e, um problema fundamental é a necessidade de manter estas cópias com a mesma informação, impedindo que um nodo acesse dados obsoletos. Existem duas estratégias básicas para manutenção da coerência de réplicas:

- *Escrita-invalidação*: uma escrita em um dado compartilhado causa a invalidação de todas as suas cópias, que passam a ser inacessíveis. Este protocolo é mais recomendado quando o programa apresenta a propriedade de localidade de acessos (*locacy of reference*) [TAN95] aos dados por processador, pois menos mensagens de notificação serão enviadas e menos cópias deverão ser atualizadas. É o protocolo mais utilizado;

- *Escrita-atualização*: uma escrita em um dado compartilhado causa a atualização de todas as suas cópias. É uma abordagem mais complicada, pois um novo valor deve ser enviado ao invés de uma mensagem de notificação, o que costuma gerar

muito tráfego na rede.

Usando estas estratégias os projetistas de sistema devem escolher o algoritmo de MCD que melhor se adapte à configuração do sistema e às características de referências de memória em aplicações típicas. Os algoritmos para implementação de MCD são divididos em três classes descritas a seguir.

2.4.1 - Algoritmos Único Leitor / Único Escritor (SRSW)

Esta classe de algoritmos proíbe replicação, enquanto permite, mas não requer migração. O algoritmo SRSW de gerenciamento de MCD mais simples é o algoritmo servidor central. A proposta consiste em um único servidor central que serve todas as requisições de acesso aos dados compartilhados, fisicamente localizado neste nodo. Este algoritmo sofre de problemas de desempenho devido ao servidor central poder tornar-se um gargalo do sistema. Tal organização implica em não distribuição dos dados compartilhados. Uma possível modificação é a distribuição estática de responsabilidade de partes do espaço de endereçamento compartilhado em diferentes servidores. Funções de mapeamento simples, como *hashing*, podem servir para localizar o servidor apropriado para o correspondente dado.

Alguns algoritmos SRSW permitem migração. Entretanto, somente uma única cópia de determinado bloco de dados pode existir, sendo que esta cópia pode migrar sob demanda. Se uma aplicação possui a propriedade de localidade de referência, o custo da migração é amortizado por vários acessos. O algoritmo pode apresentar um melhor desempenho quando uma longa seqüência de acessos for realizada neste bloco de dados por um processador, sem que outro processador o faça.

De qualquer forma, este tipo de algoritmo é pouco utilizado devido ao seu desempenho ser normalmente baixo.

2.4.2 - Algoritmos Vários Leitores / Único Escritor (MRSW)

A maior intenção dos algoritmos MRSW (também conhecidos por replicação de leitura) é reduzir o custo médio das operações de leitura, contando que compartilhamento de leitura é o padrão que prevalece em aplicações paralelas. Com esta finalidade, eles permitem operações de leitura simultâneas em execuções locais em vários nodos. Somente um nodo por vez pode receber permissão para atualizar a cópia replicada. Sendo que uma escrita nesta cópia aumenta o custo desta operação, porque os usuários das outras cópias deste dado devem ser prevenidos. Por isso, os algoritmos MRSW são usualmente baseados em invalidação [LI89].

2.4.3 - Algoritmos Vários Leitores / Vários Escritores (MRMW)

Os algoritmos MRMW (também chamados de replicação total) permitem replicação dos blocos de dados com permissão de leitura e gravação. Para preservar a coerência, alterações em cada cópia devem ser distribuídas para todas as outras cópias em nodos remotos, por mensagens *multicast* ou *broadcast*. Devido a estes algoritmos tentarem minimizar o custo de acesso das escritas, ele é apropriado para compartilhamento de escrita e, freqüentemente, funciona com protocolos de escrita-atualização. Estes algoritmos podem produzir tráfego de coerência alto, especialmente quando a frequência de atualizações e o número de cópias replicados são altos.

2.5 - Modelos de Consistência de Memória

O problema de manter uma consistência de memória implica na existência de uma política que determina como e quando mudanças feitas por um processador são vistas pelos outros processadores do sistema. A escolha do modelo de consistência define o comportamento pretendido do mecanismo de MCD, com respeito as operações de leitura e escrita. O modelo mais intuitivo de coerência de memória é o da *consistência estrita*, na qual uma operação de leitura retorna o valor de escrita mais recente. Este tipo de coerência é alcançado somente quando existe uma noção de tempo global que possa fornecer uma ordem determinística para todas as leituras e escritas. Entretanto, "escrita mais recente" é um conceito ambíguo em sistemas distribuídos, onde não existe um relógio global. Por este motivo, e também para aumentar o desempenho, foram desenvolvidos outros modelos para manter a coerência em sistemas MCD [LO 94].

Diferentes aplicações paralelas exigem diferentes modelos de consistência. Quanto mais restrito for o modelo de consistência de memória, mais ele influenciará no desempenho do sistema executando estas aplicações. Modelos de consistência rígidos, tipicamente aumentam a latência de acesso à memória e a exigência de largura de banda, enquanto simplificam a programação. Ao contrário, os modelos mais relaxados, permitem reordenação de memória, *pipelining* e *overlaping*, consequentemente aumentando o desempenho. Entretanto, exigem alto envolvimento do programador na sincronização dos acessos aos dados compartilhados.

No modelo de *consistência seqüencial*, o resultado de qualquer execução paralela é o mesmo que seria obtido se as operações de processadores individuais fossem executadas em alguma ordem seqüencial [Lamport79]. Os protocolos de

coerência de memória têm que assegurar que todos os nodos vejam a mesma seqüência de leituras e escritas. Então, o efeito de cada acesso à memória deve ser executado globalmente antes que o próximo acesso tenha permissão para executar. Isto pode ser alcançado serializando todas as leituras e escritas através de um nodo central ou com a utilização de protocolos mais eficientes.

O modelo de *consistência causal* (Hutto e Ahmad, 1990) representa um enfraquecimento da consistência seqüencial no que ela faz uma distinção entre eventos que são causalmente relacionados e aqueles que não são. Quando há uma leitura seguida depois de uma gravação, os dois eventos estão potencialmente relacionados causalmente. De maneira parecida, uma leitura está causalmente relacionada com a gravação que forneceu os dados que a leitura obteve.

Implementar uma consistência causal exige que se localize quais processos viram quais gravações. Isto significa, que um gráfico de dependência, de que operação é dependente de que outras operações, deve ser construído e mantido. Fazer isto envolve algum ônus [TAN95].

O modelo de *consistência de processador* garante que escritas de um dado processador sejam realizadas na ordem em que foram requisitadas. Entretanto, a ordem na qual escritas de dois processadores distintos são observadas por um terceiro processador não precisa ser a mesma. Em outras palavras, consistência em escritas é garantida em cada processador, mas a ordem de leituras de cada processador não é restringida, contanto que elas não envolvam outros processadores. Tal situação ocorre quando atrasos nas passagens de mensagens acarretam que uma escrita local seja vista antes que uma escrita remota, em um ambiente no qual é garantido que as mensagens são recebidas na ordem do envio. Então, se dois nodos escrevem diferentes valores em

determinada variável, cada nodo irá observar primeiro o valor de sua escrita local, para depois observar o valor da escrita realizada pelo outro nodo. Entretanto, a seqüência de escritas realizadas por qualquer um dos nodos será vista na mesma ordem por todos os outros nodos. A chave do ganho de desempenho alcançado pelo modelo de consistência de processador comparado com a consistência seqüencial é que leituras são diferenciadas das escritas.

Sistemas MCD que utilizam modelos de *consistência de memória relaxada* reduzem a freqüência na qual os dados são compartilhadas com outros processadores. Estes modelos utilizam a noção de variáveis de sincronização. Dados compartilhados são transferidos somente durante a sincronização, resultando em muito menos *overhead.* Entre as variáveis de sincronização, a ordem das operações de memória não são impostas, enquanto que as variáveis de sincronização tem que seguir as regras de consistência seqüencial.

O modelo de *consistência fraca* (*weak consistency*) distingue acessos normais (leituras e escritas) de operações especiais de sincronização. Nos pontos de sincronização é que o sistema torna-se globalmente consistente. Antes que uma operação de sincronização possa executar, todos os acessos normais anteriores devem ser completados. Além disso, acessos realizados após a operação de sincronização devem esperar para que todas as operações de sincronização sejam finalizadas. Finalmente, 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 de liberação (*release consistency*) é uma extensão do modelo de consistência fraca, que também distingue acessos normais de acessos de

sincronização. Acessos de sincronização são divididos em operações de obtenção (*acquire*) e liberação (*release*). Um sistema possui consistência de liberação se obedecer as seguintes regras:

1. todas as operações de obtenção efetuadas por um dado processador precisam ter sido completadas antes que acessos normais neste processador tenham permissão para executar;

2. todos os acessos normais efetuados por um determinado processador precisam ter sido completados antes que uma operação de liberação tenha permissão para executar;

3. acessos de sincronização precisam ter consistência de processador.

A ordenação é imposta somente nos pontos de liberação (a execução não procede além do ponto de liberação até que todas as operações de memória a variáveis compartilhadas sejam executadas). Este modelo assume que acessos conflitantes de leitura e atualização da memória são sempre protegidos utilizando mecanismos que garantam exclusão mútua, como os semáforos(que, por sua vez exigem uma implementação por passagem de mensagens em sistemas distribuídos).

Consistência de liberação preguiçosa (*lazy release consistency*) impõe ordenamento de acessos somente no próximo ponto de obtenção. Isto significa que somente o processo que requisita acesso exclusivo ao dado em processo de liberação deve esperar.

O modelo de *consistência de entrada* requer que o programador introduza novas variáveis de sincronização para proteger estruturas de dados específicas ou até mesmo importantes variáveis simples. Consequentemente, os dados são passados para o

próximo processo somente quando necessário, o que significa uma potencial melhora de desempenho mas com grande esforço de programação.

A seguir apresentamos uma tabela com o resumo das principais formas de coerência, dividindo-as em consistência sem operação de sincronização (Tabela 2.5 (a)) e com operação de sincronização (Tabela 2.5(b)) [TAN95].

Um desempenho satisfatório de um sistema MCD, depende muito do tipo de consistência de memória utilizado mas, também, do tamanho da unidade de dado a ser compartilhada entre os processos. Isto implica no volume de dados a trafegar pela rede.

Na seção seguinte, apresentamos como os dados, a serem compartilhados, podem ser organizados.

Consistência	Descrição
Estrita	Ordenação temporal absoluta de todos os eventos do acesso compartilhado
Seqüencial	Todos os processos vêem todos os acessos compartilhados na mesma ordem
Causal	Todos os processos vêem todos os acessos compartilhados relacionados por causalidade na mesma ordem
PRAM	Todos os processos vêem suas gravações mutuamente na ordem em que elas são produzidas. As gravações de processadores diferentes nem sempre podem ser vistas na mesma ordem
Processador	Consistência PRAM + coerência de memória

Tabela 2.5 (a) - Modelos de consistência que não usam operações de sincronização.

Consistência	Descrição
Fraca	Os dados compartilhados só podem ser consistentes após sincronização
Liberação	Os dados compartilhados se tornam consistentes quando uma região crítica é deixada
Entrada	Os dados compartilhados pertencentes a uma região crítica são tornados consistentes quando se entra numa região crítica

Tabela 2.5 (b) - Modelos de consistência com operações de sincronização.

2.6 - Granularidade

A estrutura dos dados representa a disposição global do espaço de endereçamento compartilhado, bem como a organização dos dados neste espaço. Estes dados podem estar organizados de diferentes maneiras. Na variação mais simples, o espaço de endereçamento é dividido em páginas, sendo que cada página estará presente na memória de uma máquina. Quando um processo acessa dados deste espaço de endereçamento compartilhado, um gerenciador de mapeamento mapea o endereço para a memória física.

Outra proposta é não compartilhar todo o espaço de endereçamento, mas somente uma parte dele, apenas aquelas variáveis ou estruturas de dados que precisam ser usadas por mais de um processo. Esta técnica reduz a quantidade de dados que devem ser compartilhados. Uma otimização possível é replicar as variáveis compartilhadas em várias máquinas. O problema então é como manter múltiplas cópias de um conjunto de estruturas de dados consistentes. Potencialmente, leituras podem ser feitas localmente sem nenhum tráfego de rede, e escritas podem ser realizadas utilizando um protocolo de atualização.

É possível estruturar ainda mais o espaço de endereçamento, ao invés de apenas compartilhar variáveis, pode-se encapsular tipos de dados (objetos). Esta proposta difere da proposta de variáveis compartilhadas no ponto em que cada objeto tem não apenas alguns dados, mas também procedimentos (métodos). Processos podem apenas manipular os dados do objeto invocando seus métodos. Acesso direto aos dados não são permitidos. Restringindo o acesso desta forma, novas otimizações tornam-se possíveis.

Os sistemas MCD são parecidos aos multiprocessadores em certos aspectos. Em ambos os sistemas, quando uma palavra de memória não local é consultada, uma porção da memória contendo a palavra é solicitada de seu local atual e colocada na máquina que realiza a consulta (memória principal ou *cache*, respectivamente). Uma importante questão de projeto é de que tamanho deve ser a porção. As possibilidades são a palavra, bloco (algumas palavras) , página ou segmento (páginas múltiplas).

Com um multiprocessador, solicitar uma única palavra ou algumas poucas dezenas de bytes é plausível porque a MMU sabe exatamente qual endereço foi consultado e a hora de ajustar uma transferência com barramento é medida em *nanossegundos*. Memnet [DEL91], embora não estritamente um multiprocessador, também usa um tamanho de porção pequeno (32 bytes). Com os sistemas MCD, tal granulosidade fina é difícil ou impossível, devido ao modo como a MMU trabalha.

Quando um processo consulta em uma palavra que se encontra ausente, há uma falha na página. Uma escolha óbvia é trazer a página inteira de que se precisa. Além do mais, integrar MCD com memória virtual torna o projeto total mais simples, já que a mesma unidade, a página, é usada por ambas. Numa falta de página (*page fault*), a página ausente é trazida de outra máquina em vez do disco, tanto que muito do código

de manuseio de falta de página é o mesmo do caso tradicional.

Entretanto, outra escolha possível é trazer uma unidade maior, com um região de 2, 4 ou 8 páginas, incluindo a página de que se precisa. Porém, fazer isso simula uma página de tamanho maior. Há vantagens e desvantagens de uma porção de tamanho maior para a MCD. A maior vantagem é que devido ao tempo de início de uma transferência através de rede ser substancial, não leva muito mais tempo transferir 1024 *bytes* do que 512 *bytes*. Transferindo-se dados em unidades grandes, quando um pedaço grande de um espaço de endereço tem que ser movido, o número de transferências pode muitas vezes ser reduzido. Esta propriedade é importante especialmente porque muitos programas exibem localidade de consulta, o que significa que se um programa consultou uma palavra numa página, possivelmente irá consultar outras palavras da mesma página num futuro imediato.

Por outro lado, a rede ficará amarrada mais tempo com uma transferência maior, bloqueando outras faltas causadas por outros processos. Também, um tamanho efetivo de página maior introduz um problema novo, chamado de *compartilhamento falso* [TAN95]. O compartilhamento falso acontece quando uma página compartilhada, possui mais informações do que as efetivamente compartilhadas, causando desperdício e mais congestionamento na rede.

2.7 - Modelos de sincronização distribuída

Muitas aplicações fazem verificações relativas aos valores armazenados em memória compartilhada. Isto é válido para aplicações baseadas em MCD, como aplicações escritas para multiprocessadores de memória compartilhada (ou realmente

para qualquer programa concorrente que compartilha dados, como núcleos de sistema operacional e servidores *multi-threads*). A solução é fragmentar a aplicação, criando uma seção crítica para sincronizar processos e, assegurar que a atualização de variáveis compartilhadas seja executada por somente um processo de cada vez (exclusão mútua).

Portanto, um serviço de sincronização distribuído precisa ser provido de *locks* e semáforos. Até mesmo quando MCD é estruturado como um conjunto de objetos, os implementadores dos objetos têm que utilizar sincronização. Sincronização é implementada usando passagem de mensagem. Instruções de máquina especiais como *test-and-set* que é usado para sincronização em multiprocessadores de memória compartilhado são em princípio aplicáveis à MCD baseada em página, mas a operação delas no caso distribuído é muito ineficiente. As mais recentes implementações de MCD, tiram proveito de sincronização de aplicação para reduzir o número de transmissões de atualização. A MCD inclui sincronização então como um componente integrado.

CAPÍTULO III

SISTEMAS DE MCD

O nível onde o mecanismo de MCD é implementado é uma das decisões mais importantes na construção de sistemas MCD, afetando o custo de programação e o desempenho global do sistema [PRO96].

Para alcançar facilidade de programação, custo efetivo e escalabilidade, sistemas MCD implementam o modelo de memória compartilhada em memória fisicamente distribuída. Devido a MCD distribuir o espaço de endereçamento compartilhado através das memórias locais, pesquisas devem ser executadas em cada acesso a estes dados, para determinar se o dado requisitado está na memória local. Senão, o sistema deve trazer o dado para a memória local. O sistema também deve realizar uma ação em acessos de escrita para preservar a coerência dos dados compartilhados. Tanto pesquisas quanto ações podem executar em software, hardware ou de forma combinada. De acordo com esta propriedade, os sistemas podem ser classificados em três grupos: software, hardware e implementações híbridas.

A escolha da implementação depende da relação preço/desempenho. Embora, tipicamente superior em desempenho, implementações em hardware requerem

características controladas por software, as quais são explicitamente realizadas pelo programador com o objetivo de otimizar as referências a memória. Também várias propostas de MCD por software exigem algum suporte de hardware. Portanto, torna-se mais natural empregar métodos híbridos, com elementos de software e hardware combinados para balancear a relação de custo e complexidade.

A gerência da hierarquia de memória é um aspecto que requer cuidados, no sentido de encontrar algoritmos eficientes para mover os dados dinamicamente entre os diferentes níveis da memória ou níveis de cache. Um problema é como mapear as estruturas de dados do espaço de endereçamento lógico compartilhado em módulos de memória fisicamente distribuídos. Porções do espaço de memória lógica são mapeados unicamente na memória física (uma porção lógica mapeada para uma localização física de mesmo tamanho) como nas máquinas CC-NUMA (*cache-coherent non-uniform memory access*). Outra possibilidade é utilizar replicação (uma porção lógica mapeada para várias localizações físicas, cada uma do mesmo tamanho que a porção lógica) como nas máquinas COMA (*cache-only memory architectures*) e máquinas RMS (*reflective memory systems*) [MIL99]. Portanto, de acordo com a arquitetura do sistema de memória, três grupos de sistemas de hardware MCD são especialmente interessantes:

- CC-NUMA – sem acesso uniforme à memória, com coerência de *cache*;
- COMA – arquitetura de memória somente *cache*;
- RMS – sistemas de memória refletida.

3.1.1 - Sistemas de MCD CC-NUMA

Um sistema CC-NUMA (Figura 3.1) distribui estaticamente o espaço de

endereçamento virtual compartilhado através da memória local dos *clusters*. Tanto o processador local como processadores de outros *clusters* do sistema podem acessar este espaço de endereçamento, embora com diferente latência de acesso. Ou seja, processadores de um mesmo *cluster* terão acesso a memória local deste *cluster* na velocidade do hardware. No entanto, acessos provenientes de processadores de outros *clusters* terão adicionado a latência de acesso a memória a latência de comunicação.

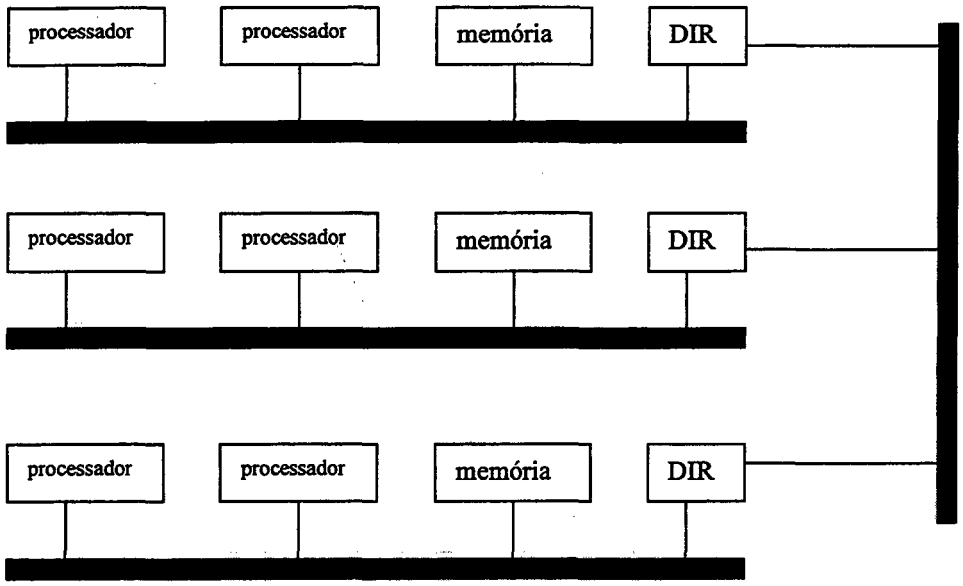

Figura 3.1 – Arquitetura de memória CC-NUMA

O mecanismo de MCD é normalmente implementado utilizando diretórios com organização variando de mapeamento completo à diferentes estruturas dinâmicas, como as listas simples ou duplamente encadeadas e as árvores. O principal esforço é alcançar alto desempenho (como em esquemas de mapeamento completo) e boa escalabilidade fornecida pela redução do *overhead* de armazenar o diretório. Para minimizar a latência, o particionamento estático de dados pode ser feito cuidadosamente, para maximizar a freqüência de acessos locais. Indicadores de

desempenho também dependem altamente da topologia de interconexão. O mecanismo de invalidação é tipicamente aplicado para fornecer consistência, enquanto alguns modelos de consistência de memória relaxados podem servir como fonte de ganho de desempenho. Exemplos típicos desta proposta são Memnet, Dash e SCI.

3.1.1.1 - Memnet

O multiprocessador baseado em anel – Memnet (Memory as Network Abstraction) – foi um dos primeiros sistemas MCD implementados em hardware [DEL91]. O principal objetivo deste sistema é evitar o custo de comunicação entre processos via passagem de mensagens e fornecer a abstração de memória compartilhada para uma aplicação diretamente pela rede, sem intervenção do *kernel* do sistema operacional. O espaço de endereçamento do Memnet é mapeado na memória local de cada *cluster* (uma área reservada) em padrão NUMA. Outra parte de cada memória local é a área de cache, a qual é utilizada para replicação dos blocos de 32-bytes cuja área reservada está em algum nodo remoto. O protocolo de coerência é implementado em hardware, através de uma máquina de estado contida no dispositivo Memnet em cada *cluster* – um controlador duplo de memória, um para seu barramento local e um para a interface do anel.

No caso de uma falta de página memória local, o dispositivo Memnet envia uma mensagem apropriada, a qual circula no anel. Cada um destes dispositivos do anel examina a mensagem (*snooping*). O *cluster* mais próximo com uma cópia válida satisfaz a requisição inserindo o dado requisitado na mensagem antes de reenviá-la. Uma requisição de escrita para uma cópia não exclusiva resulta em uma mensagem que

invalida as outras cópias compartilhadas. Cada dispositivo Memnet tendo uma cópia válida daquele bloco recebe uma mensagem de invalidação. Finalmente, a interface do *cluster* que gera a mensagem recebe e remove-a do anel.

3.1.1.2 - Dash

Dash (Directory Architecture for Shared Memory) – Arquitetura de diretório para memória compartilhada, é uma arquitetura escalável de *cluster* de multiprocessadores utilizando um mecanismo de hardware MCD baseado em diretórios [LEN92]. Cada *cluster* de quatro processadores contém uma igual parte da memória compartilhada total do sistema (domínio do *home*) e as entradas de diretório correspondentes. Cada processador também tem uma cache privada em dois níveis de hierarquia onde a localização da memória (remota) de outros *cluster* pode ser replicada ou migrada em blocos de 16-bytes. A hierarquia de memória do Dash é desdobrada em quatro níveis:

- Cache do processador;
- Caches de outros processadores no *cluster* local;
- *Cluster home* (o *cluster* que contém diretório e memória física para dado bloco de memória);
- *Cluster* remoto (o *cluster* marcado pelo diretório como mantendo uma cópia do bloco).

Manutenção da coerência é baseada em um protocolo de mapeamento completo de diretório. Um bloco de memória pode estar em um dos três estados:

uncached (não está presente em nenhuma cache fora do *cluster home*), *cached* (existem uma ou mais cópias inalteradas em *clusters* remotos) ou *dirty* (modificado em algum *cluster* remoto). Usualmente, devido a propriedade de localidade, referências podem ser satisfeitas no *cluster*. De outra forma, a requisição vai para o *cluster home* do bloco envolvido, o qual realiza alguma ação de acordo com o estado encontrado no seu diretório. Um modelo de consistência de memória relaxado – consistência de liberação – aumenta o desempenho e faz otimizações no acesso a memória. Técnicas para reduzir a latência de memória, tais como *prefetching* controlados por software, operações de alteração e de transferência também aumentam o desempenho. Dash fornece suporte de hardware para sincronização.

3.1.2 - Sistemas de MCD COMA

A arquitetura COMA (Figura 3.2) usa a memória local dos *clusters* como grandes caches para blocos de dados do espaço de endereçamento virtual compartilhado (memórias de interesse). Não existe uma localização *home* pré-determinada na memória física para um item de dado em particular, e eles podem ser replicados e migrados nas memórias de interesse sob demanda. Por conseguinte, a distribuição de dados nas memórias locais e caches adaptam-se dinamicamente ao comportamento da aplicação.

Arquiteturas COMA têm duas topologias de rede hierárquicas que simplificam os dois principais problemas deste tipo de sistemas: localizar um bloco de dados e substituí-lo. Elas são menos sensíveis à distribuição estática de dados do que são as arquiteturas NUMA. Devido à organização de suas caches, as memórias de interesse reduzem o custo entre volume e conflitos de ausência. Mas, a estrutura

hierárquica impõe latência de comunicação e de falha de dados ligeiramente mais alta. É inerente a estas arquiteturas um aumento no *overhead* de armazenamento para manter informações típicas para memórias cache. Os sistemas mais relevantes são KSR1 e DDM.

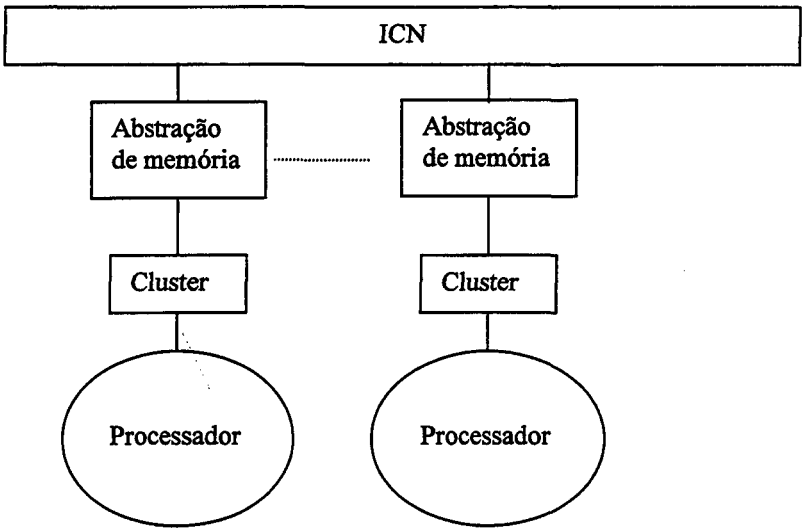

Figura 3.2 – Arquitetura de memória COMA

3.1.2.1 - KSR1

O multiprocessador KSR1 representa um dos precursores preocupados em fazer um sistema MCD disponível no mercado [FRA93]. Ele consiste de uma organização hierárquica de *cluster* baseada em anel, cada um com cache local de 32 *Mbytes*. A unidade de alocação nas caches locais é uma página (16 Kbytes), enquanto que a unidade de transferência e compartilhamento nas caches locais é uma sub-página (128 bytes).

O hardware dedicado responsável por localizar, copiar e manter a coerência das sub-páginas nas caches locais é chamada máquina Allcache, a qual é organizada

como uma hierarquia com diretórios em níveis intermediários. Esta máquina transparentemente determina o caminho das requisições através da hierarquia. Acessos mal sucedidos são provavelmente satisfeitos por *clusters* na mesmo nível hierárquico ou próximo a ele. Deste modo, a organização do Allcache minimiza o caminho para localizar um endereço particular.

O protocolo de coerência é baseado em invalidação. Estados possíveis de uma sub-página em uma cache local particular são: exclusivo (apenas uma cópia válida), *nonexclusive* (proprietário; várias cópias existem), cópia (não proprietário; cópia válida) e inválido (não válida, mas sub-página alocada). Além deste estados, KSR1 fornece estados atômicos para sincronização. Operações de *locking* e *unlocking* nas sub-páginas são alcançadas com instruções especiais. Como em todas as arquiteturas sem memória principal onde todos os dados são armazenados em caches, surge o problema de substituição das linhas de cache. Não existe um destino padrão para a linha na memória principal, então a escolha do novo destino e a alteração do diretório podem ser complicados e consumirem tempo. Além disso, propagações de requisições através dos diretórios hierárquicos causam longas latências.

3.1.2.2 - DDM

O protótipo da DDM (Data Diffusion Machine) é feito de *clusters* de quatro processadores com memórias de interesse e um barramento assíncrono por dividir transações [HAG92]. Anexando um diretório no topo do barramento DDM local, habilita sua comunicação com barramentos do mesmo tipo de um nível superior, permite um grande sistema baseado em hierarquias de diretório e barramento (oposto a

hierarquia baseada em anel do KSR1). O diretório é uma memória associativa que armazena a informação de estado para todos os itens na memória de interesse abaixo dele, mas sem os dados.

Um protocolo de coerência *snoopy* escrita-invalidação trata do interesse dos dados na leitura, apaga os dados replicados na escrita, e gerencia a substituição quando um conjunto na memória de interesse está cheio. Um item pode estar em sete estados. Três correspondem à inválido, exclusivo e válido, típicos de protocolos *snoopy*. Os quatro restantes são estados transientes necessários para lembras as pendências no barramento. Transações que não podem ser completadas em um nível mais baixo vão através do diretório ao nível acima. Similarmente, o diretório reconhece as transações que precisam ser servidas por um sub-sistema e encaminham-nas para um nível abaixo.

3.1.3 - Sistemas MCD de Memória Refletida

Sistemas de memória refletida têm um mecanismo implementado em hardware para atualização dados de granulosidade fina. O espaço de endereçamento global compartilhado é formado fora dos segmentos de memória local. Estes segmentos são designados como compartilhados e mapeados para este espaço através de tabelas de mapeamento programáveis presente em cada *cluster* (Figura 3.3). Portanto, as partes deste espaço compartilhado são seletivamente replicados (refletidos) ao longo de diferentes *clusters*. Manutenção da coerência das regiões compartilhadas é baseada no algoritmo de replicação total (MRMW). Para manter os dados atualizados, cada escrita para um endereço contido neste espaço de endereçamento compartilhado no *cluster* propaga-se através de um *broadcast* ou de um *multicast* para todos os outros *clusters* onde o mesmo endereço estiver mapeado.

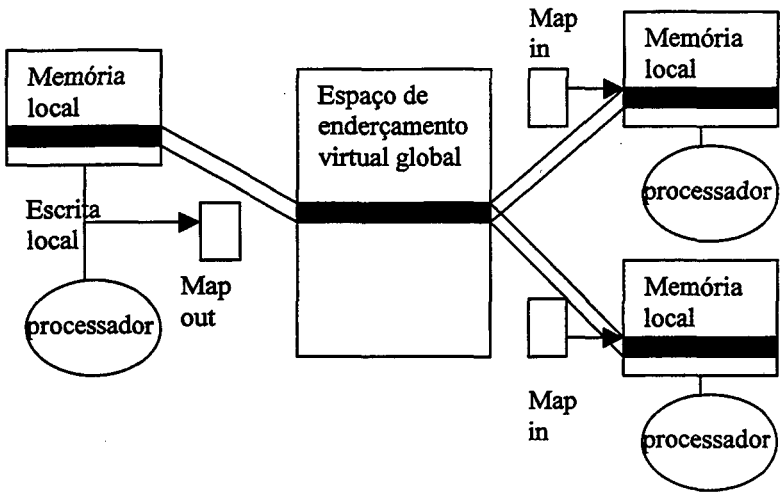

Figura 3.3 – Arquitetura MCD de memória refletida

O processador não protela escritas nem computações sobrepostas com comunicação. Isto é uma fonte de melhoria de desempenho típica de modelos de consistência de memória relaxada. Também não existe contenção e longas latências como em típicos sistemas de memória compartilhada, devido ao acesso irrestrito aos dados compartilhados e acessos simultâneos á cópias locais são garantidas. Mas, todas as leituras de memória compartilhada são locais, com tempo de acesso determinístico. O princípio deste mecanismo de MCD assemelha-se aos protocolos de coerência de cache escrita-atualização.

3.1.3.1 - RMS

Muitos sistemas com diferentes *clusters* e topologias de rede aplicam memória refletida. Devido ao *broadcast* ser o mecanismo mais apropriado para alterar os segmentos replicados, a topologia do barramento compartilhado é importante na

arquitetura destes sistemas. A Encore Computer Corporation desenvolveu vários RMSs baseados em barramentos para uma grande quantidade de aplicações [LUC95].

Estes sistemas normalmente consistem de um pequeno número de *clusters* de minicomputadores interconectados por um barramento RM, um barramento apenas-escrita. O tráfego consiste somente da transferência de palavras de dados (endereço + valor da palavra de dados). Foram desenvolvidas melhorias, e o canal de memória também permite alterações baseada em blocos. A unidade de replicação é um segmento de 8 Kbytes. Segmentos são tratadas como janelas que podem ser abertas (mapeadas no espaço de compartilhamento refletido) ou fechada (desabilitada para reflexão e exclusivamente acessada por cada *cluster* em particular). Um segmento replicado pode apontar para diferentes endereços em cada *cluster*. Portanto, tabelas de translação são fornecidas separadamente para o lado da transmissão (para cada bloco de memória local) e da recepção (para cada bloco do espaço de endereçamento refletido).

3.2 - Implementações de MCD por Software

A idéia de construir um mecanismo de software que forneça um paradigma de memória compartilhada para o programador, pode ser alcançado em nível do usuário, rotinas de bibliotecas em tempo de execução, sistema operacional ou linguagem de programação.

Granulosidade grossa(na ordem de *kilobytes*) é utilizada normalmente em soluções de software, devido ao gerenciamento de MCD ser usualmente suportado através da memória virtual. Assim, se o dado requisitado está ausente na memória local, a página será buscada ou da memória local de outro *cluster* ou do disco. Páginas com

tamanho grande são vantajosas para aplicações que apresentam localidade de referências, e também reduzem o espaço necessário para o armazenamento de diretório. Mas, programas paralelos caracterizados com granulosidade fina em nível de compartilhamento são afetados, devido a falsos compartilhamentos ou desperdícios.

Suportes de MCD por software geralmente são mais flexíveis do que suportes por hardware, e habilitam um melhor condicionamento dos mecanismos de consistência ao comportamento da aplicação. Entretanto, normalmente não podem competir com as implementações em hardware em termos de desempenho. Uma vez que não utilizam aceleradores de hardware para resolver o problema, projetistas elaboraram modelos de consistência relaxado, embora isso acarrete em maior trabalho para o programador. Devido as pesquisas serem realizadas em uma grande quantidade de linguagens de programação e sistemas operacionais disponíveis, numerosas implementações de MCD por software foram desenvolvidas. A seguir são descritas algumas dessas implementações.

3.2.1 - Treadmarks

Treadmarks [ZWA94], [ZWA96] suporta computação paralela em redes de *workstations*. Sua principal característica é que provê um espaço de endereçamento compartilhado global para as diferentes máquinas em um *cluster*. TreadMarks incorpora várias características, inclusive consistência de lançamento e protocolos de escritores múltiplos.

O projeto TreadMarks estuda a integração de compilador e técnicas de *runtime*, o uso de multithreading, em particular em nodos de multiprocessador,

suportando espaços de endereçamento grandes, heterogeneidade, e escalabilidade. TreadMarks é compatível com IBM, DEC, Solaris, HP, x86 (FreeBSD 2.1R concorrente ou Linux Slackware 3.0) e hardware de SGI. Um projeto para WindowsNT também foi completado. São suportados as linguagens C, C++, JAVA, e FORTRAN.

A memória compartilhada é implementada como uma seqüência linear de bytes, baseado no modelo de consistência *release*. A implementação de TreadMarks usa o hardware da memória virtual para detectar acessos, bem como o protocolo de múltipla-escrita para diminuir problemas causados por diferentes tamanhos de página de memória e granulosidade das aplicações.

TreadMarks é executado no nível de usuário em redes Unix de estações de trabalho, sem modificações no *kernel* ou privilégios especiais, juntamente com a interface padrão e compiladores do Unix. Por esse motivo, o sistema é largamente portável para várias plataformas.

Para a comunicação entre as estações de trabalho o sistema utiliza UDP/IP através da interface de *sockets* da Berkeley. Quando um nodo envia uma mensagem de requisição, fica bloqueado até que a mensagem de retorno chegue.

O protocolo de coerência é implementado por uma chamada de sistema que controla o acesso às páginas compartilhadas. Qualquer tentativa de adquirir acesso restrito para uma página compartilhada gera um SIGSEGV *signal*. A rotina examina a estrutura de dados local para determinar o estado da página e examina a pilha de exceções para determinar se a referência é de leitura ou escrita. Se a referência é uma leitura, a proteção da página é alterada para leitura. Para um operação de escrita, é criado uma cópia das páginas livres, da mesma forma que ocorre em resposta para uma

falha causada por uma escrita para uma página em modo somente para leitura. A seguir o manipulador atualiza os direitos de acesso para a página original e retorna.

3.2.2 - Munin

O sistema Munin é fundamentado no modelo MCD por objetos de *software*. Entretanto, pode também colocar cada objeto numa página separada para que o hardware MMU gerencie os acessos aos objetos compartilhados (Bennett et al., 1990; e Carter et al., 1991, 1993). O modelo básico usado por Munin é o de múltiplos processadores, cada um com um espaço de endereço linear paginado em que uma ou mais *threads* estão rodando um programa de multiprocessador ligeiramente modificado. O objetivo do projeto Munin é pegar programas de multiprocessador já existentes, fazer pequenas mudanças neles e fazê-los rodar eficientemente em sistemas de multicomputadores usando uma forma de MCD. Obtém-se uma bom desempenho com uma variedade de técnicas a serem descritas adiante, incluindo o uso da consistência de liberação em vez da consistência seqüencial.

As mudanças consistem em se anotar as declarações das variáveis compartilhadas com a palavra-chave *compartilhada*, de tal maneira que o compilador possa reconhecê-las. As informações sobre o padrão de uso esperado também podem ser fornecidas, para se permitir que certos casos especiais sejam reconhecidos e otimizados. Por padrão, o compilador põe cada variável compartilhada numa página separada, embora variáveis compartilhadas grandes, como vetores, possam ocupar várias páginas. Também é possível para o programador especificar que as variáveis do mesmo tipo (Munin) sejam colocadas na mesma página. Misturar os tipos não funciona, já que o

protocolo de consistência usado por uma página depende do tipo de variáveis contidas nela.

Para rodar o programa compilado, um processo privilegiado (*root*) é inicializado em um dos processadores. Este processo pode gerar novos processos em outros processadores, que então rodam em paralelo com o principal e se comunica com ele mutuamente usando variáveis compartilhadas, como programas normais de multiprocessador fazem. Uma vez inicializado um determinado processador, um processo não se move.

Os acessos a variáveis compartilhadas são feitos usando-se as instruções de leitura e gravação normais. Não são usados métodos protegidos especiais. Se for feita uma tentativa de se usar uma variável compartilhada que não está presente, ocorre uma falta de página, e o sistema Munin assume o controle.

A sincronização para a exclusão mútua é feita de modo especial e está relacionada com o modelo de consistência de memória. As variáveis de bloqueio podem ser declaradas e os procedimentos de biblioteca são fornecidos pelo *bloqueio* e *desbloqueio* delas. As barreiras, as variáveis de condição e outras variáveis de sincronização também são suportadas.

O que o Munin faz é fornecer os instrumentos para os usuários estruturarem seus programas em torno de regiões críticas, definidas dinamicamente por chamadas de aquisição (entrada) e liberação (saída). As gravações para variáveis compartilhadas devem ocorrer dentro das regiões críticas; as leituras podem ocorrer dentro ou fora. Enquanto um processo está ativo dentro de uma região crítica, o sistema não dá garantia de consistência das variáveis compartilhadas, mas quando a execução deixa

uma região crítica, as variáveis compartilhadas modificadas desde a última liberação são atualizadas em todas as máquinas. Para programas que obedecem este modelo de programação, a Memória Compartilhada Distribuída se comporta como se ela fosse seqüencialmente consistente.

O Munin distingue três classes de variáveis. As *variáveis ordinárias* não são compartilhadas e podem ser lidas e gravadas somente pelo processo que as criou. As *variáveis de dados compartilhados* são visíveis aos vários processos e aparecem seqüencialmente consistentes, já que todos os processos as usam somente em regiões críticas. Elas devem ser declaradas como tais, mas são acessadas usando instruções de leitura e gravação normais. As *variáveis de sincronização*, tais como *locks* e barreiras, são especiais e somente acessíveis via procedimentos de acesso fornecidos pelo sistema. São estes procedimentos que fazem a Memória Compartilhada Distribuída trabalhar.

Além de usar consistência de liberação, Munin também usa outras técnicas para melhorar o desempenho. A principal dentre elas é permitir que o programador faça as declarações de variável compartilhada classificando cada uma dentro de uma das quatro categorias abaixo:

1. Somente para leitura.
2. Migratória.
3. De gravação compartilhada.
4. Convencional.

Cada máquina mantém um diretório listando cada variável, dizendo, entre outras coisas, qual categoria cada uma pertence. Para cada categoria é usado um protocolo diferente.

As variáveis compartilhadas que não são anotadas como pertencentes a uma das categorias acima são tratadas como num sistema MCD baseado em páginas convencional: somente uma cópia de cada página gravável é permitida, e é movida de processo em processo conforme a demanda.

O Munin usa diretórios para localizar as páginas que contêm variáveis compartilhadas. Quando ocorre uma falha numa consulta a uma variável compartilhada, o Munin analisa o endereço virtual que causou a falha para encontrar o nome da variável no diretório das variáveis compartilhadas. A partir do nome, ele vê que de qual categoria é a variável, se há uma cópia local, e quem é o dono provável.. As páginas de gravação compartilhada não têm necessariamente um único dono. Para uma variável compartilhada convencional, é o último processo a obter o acesso à gravação. Para uma variável compartilhada migratória, o dono é o processo que a detém atualmente.

Para se reduzir o ônus de se enviar atualizações aos processos que não estão interessados em certas páginas de gravação compartilhadas, é usado um algoritmo baseado em temporizador. Se um processo detém uma página, não a consulta dentro de um certo período de tempo e recebe uma atualização, ele dispensa a página. Da próxima vez que ele receber uma atualização para uma página dispensada, o processo diz ao processo de atualização que ele não detém mais a cópia, e o atualizador reduz o tamanho do conjunto de cópias. O provável dono na seqüência é usado para denotar a cópia do último recurso, que não pode ser dispensado sem que haja um novo dono ou nova gravação em disco. Este mecanismo garante que a página não seja dispensada por todos os processos e, consequentemente, perdida.

O sistema Munin mantém um segundo diretório para variáveis de sincronização. Estas são localizadas num modo análogo ao das variáveis compartilhadas

ordinárias. Conceitualmente, as variáveis *locks* agem como se fossem centralizadas, mas na verdade, é usada uma implementação para evitar que haja muito tráfego com mensagens enviadas para outras máquinas.

Quando um processo quer adquirir uma variável de bloqueio, primeiro ele verifica se ele mesmo possui uma. Se ele possuir uma e ela estiver livre, a solicitação é respondida. Se a variável de bloqueio não for local, ela é localizada usando-se o diretório de sincronização, que encontra o dono provável. Se a variável de bloqueio estiver livre, ela é fornecida. Se não estiver livre, o solicitante vai para o final da fila. Assim, cada processo sabe a identidade do processo seguinte na fila. Quando uma variável de bloqueio é liberada, o dono a passa para o próximo processo da lista.

As barreiras são implementadas por um servidor central. Quando se cria uma barreira, é fornecido o número de processos que devem estar esperando por ela antes de todos poderem ser liberados. Quando um processo terminou certa fase em sua computação, ele pode enviar uma mensagem ao servidor da barreira pedindo que espere. Quando o número requisitado de processos estão esperando, todos eles recebem uma mensagem liberando-os.

3.2.3 - Midway

O Midway é um sistema de MCD baseado em estruturas individuais de dados compartilhados. É parecido com o Munin de algumas formas, mas tem algumas características particulares interessantes. Seu objetivo é permitir que programas já existentes e programas novos rodem eficientemente em multicomputadores com somente algumas mudanças no código.

Os programas no Midway são basicamente programas convencionais escritos em linguagem C, C++ ou ML, com certas informações adicionais fornecidas pelo programador. Os programas Midway usam o pacote Mach C-threads para expressar paralelismo. A processo do Midway é manter as variáveis compartilhadas de modo eficiente.

A consistência é mantida requisitando-se que todos os acessos às variáveis compartilhadas e estruturas de dados sejam feitos dentro de um tipo de seção crítica específica conhecida como sistema *runtime* Midway. Cada uma destas seções críticas é guardada por uma variável de sincronização, geralmente uma variável de bloqueio, mas também, possivelmente uma barreira. Cada variável compartilhada acessada numa seção crítica deve estar explicitamente associada com uma variável de bloqueio (ou barreira) daquela seção crítica por meio de uma chamada de procedimento. Assim, quando uma seção crítica é adentrada ou deixada, o Midway sabe precisamente quais variáveis compartilhadas serão potencialmente acessadas ou já foram acessadas.

O Midway suporta a consistência de entrada, que trabalha da seguinte forma: Para acessar dados compartilhados, um processo normalmente adentra uma região crítica chamando um procedimento de biblioteca, bloqueio, com uma variável de bloqueio como parâmetro. A chamada também especifica se é necessário uma variável de bloqueio exclusiva ou uma variável de bloqueio não-exclusiva. Uma variável de bloqueio exclusiva é necessária quando uma ou mais variáveis compartilhadas tem que ser atualizadas. Se as variáveis compartilhadas só deverão ser lidas, e não modificadas, uma variável de bloqueio não-exclusiva é o suficiente, o que permite que múltiplos processos adentrem uma região crítica ao mesmo tempo. Não há problemas nessa abordagem, porque nenhuma das variáveis pode ser modificada.

Quando uma variável de bloqueio é chamada, o sistema de tempo de execução Midway adquire a variável de bloqueio e, ao mesmo tempo, traz todas as variáveis compartilhadas associadas com aquela variável de bloqueio até o momento. Fazer isso exige que se envie mensagens para os outros processos para conseguir os valores mais recentes. Quando todas as respostas são recebidas, a variável de bloqueio é fornecida (supondo-se que não há nenhum conflito) e o processo começa a executar a região crítica. Quando o processo completa a seção crítica, ele libera a variável de bloqueio. Diferente da consistência de liberação, não há comunicação na hora da liberação, isto é, as variáveis compartilhadas modificadas não são enviadas para outras máquinas que usam as variáveis compartilhadas. Somente quando um dos seus processos subseqüentemente adquire uma variável de bloqueio e pede os valores atuais é que os dados são transferidos.

Para se fazer a consistência de entrada funcionar, o Midway exige que os programas tenham três características que os programas de multiprocessador não possuem:

a) as variáveis compartilhadas devem ser declaradas usando-se a nova palavra-chave *shared;*

b) cada variável compartilhada deve estar associada com uma variável de bloqueio ou barreira;

c) as variáveis compartilhadas só podem ser acessadas dentro de regiões críticas.

Executar estas tarefas exige um esforço extra do programador. Se estas regras não forem seguidas completamente, não é gerada a mensagem de erro e o

programa pode trazer resultados incorretos. Pelo fato desta forma de programação facilitar o surgimento de erros, especialmente quando executa programas de multiprocessador antigos que ninguém mais entende, o Midway também suporta a consistência seqüencial e a consistência de liberação. Estes modelos exigem menos informação detalhada e menos esforço de programação para operarem corretamente.

A informação extra exigida pelo Midway deve ser pensada como parte do contrato entre o software e a memória que nós já estudamos antes sob o tema consistência. Com efeito, se o programa concorda em rodar de acordo com certas regras conhecidas antecipadamente, a memória promete funcionar. Caso contrário, é provável que não funcione.

CAPÍTULO IV

DESCRIÇÃO DO AMBIENTE MMG

O principal objetivo do ambiente MMG é fornecer ao programador de aplicações paralelas uma maior facilidade de programação. O compartilhamento dos dados é feito criando primitivas baseadas nas chamadas de sistema do UNIX referentes ao IPC (*Interprocess Communication*) que implementam memória compartilhada, para suportar memória compartilhada distribuída.

A organização básica do ambiente é formada por um conjunto de nodos (computadores PC compatíveis) conectados por uma rede de interconexão sobre o sistema operacional Linux. A comunicação entre os processos remotos é realizada através de *sockets*.

O ambiente é baseado no modelo cliente/servidor e os serviços de memória compartilhada distribuída são oferecidos através de um servidor que integra várias funcionalidades como: gerenciador de memória compartilhada, sincronização e nomes. Um processo cliente executa uma chamada IPC modificada, que faz a localização do dado e realiza o processo de coerência de forma transparente para o programador. Ao

término da execução da requisição pelo servidor, é retornado ao processo cliente as informações solicitadas.

Este ambiente utiliza primitivas simples e conhecidas como IPC de memória compartilhada e é implementado a nível do sistema operacional, ficando este com uma funcionalidade a mais, além de permitir que programas escritos utilizando IPC possam ser executados em ambiente paralelo, bastando alterar as implementações das chamadas do IPC originais para as novas chamadas.

4.1 - Granulosidade e nível de compartilhamento de dados

O ambiente MMG fornece a abstração de memória compartilhada pela criação de um segmento de dados compartilhados no espaço de endereçamento dos gerentes de memória compartilhada distribuída (GMD) (Figura 4.1).

Este compartilhamento é feito de forma transparente para o programador, pois ele apenas tem fazer uso das primitivas do IPC modificadas ao invés das originais.

4.2 - Política de Compartilhamento de Dados

Para obter melhores desempenhos, através da programação paralela, deve-se permitir que o maior número de processos execute simultaneamente. Deve se evitar que os processos que possuem cópias locais dos dados não fiquem bloqueados esperando o término de acessos.

O ambiente MMG utiliza a técnica de replicação para tornar os dados compartilhados disponíveis para o maior número de processos. É importante lembrar

que se os acessos às variáveis compartilhadas forem para leitura, o grau de paralelismo aumenta. Mas, se algum processo realizar acesso para escrita, os demais processos que requisitarem acesso a estas variáveis ficarão bloqueados devido às operações de coerência sobre estes dados compartilhados.

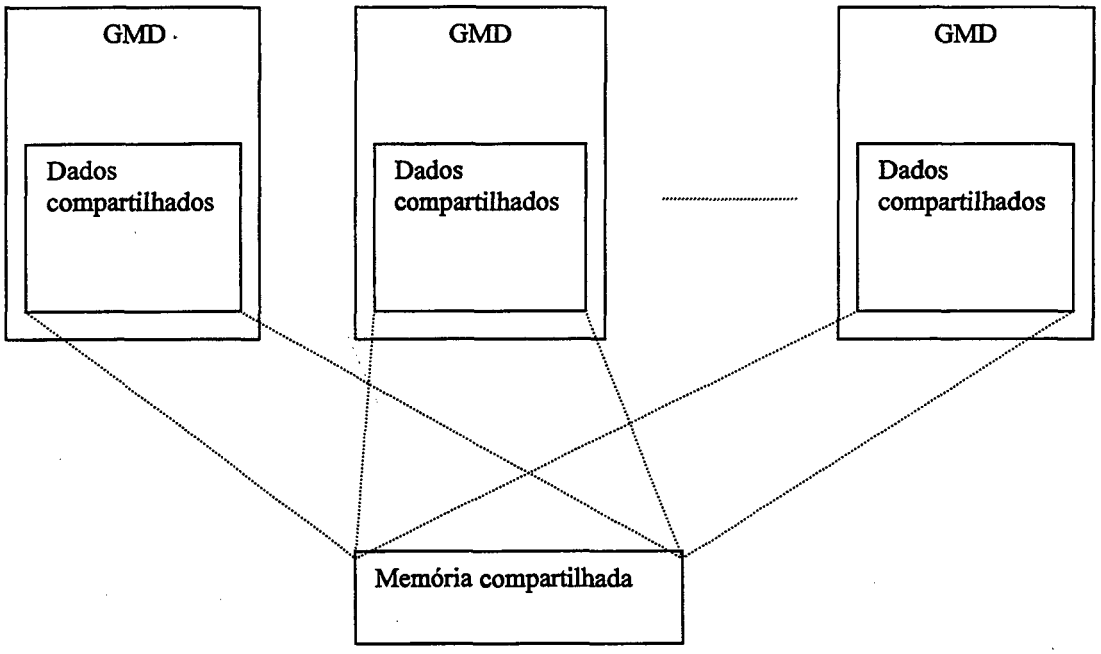

Figura 4.1 – Compartilhamento de dados no ambiente MMG.

Apesar da técnica de replicação aumentar a complexidade do ambiente para manter as cópias coerentes, sua utilização é justificada pelo ganho de desempenho nos sistemas de memória compartilhada distribuída [LI89].

4.3 - Semântica de Coerência

As semânticas que possibilitam uma programação mais fácil são semânticas

mais restritivas, portanto não apresentam um desempenho tão bom como as semânticas mais leves e de difícil programação.

O ambiente MMG permite a utilização de várias semânticas de coerência para o programador: seqüencial, processador e entrada. O programador pode optar por uma das três semânticas e utilizar a que melhor se adaptar à sua aplicação.

4.4 - Política de Coerência

O ambiente MMG utiliza o protocolo de coerência escrita-atualização. Quando um processo altera o valor de um dado compartilhado, as cópias deste dado que se encontram em outros nodos são atualizadas. Este tipo de protocolo pode causar um desperdício de tráfego na rede, pois um processo que possui a cópia recentemente atualiza, pode não mais requisitar o dado. Entretanto, sempre que um processo requisitar o dado compartilhado e este tiver uma cópia local válida, esta pode ser acessada diretamente sem ter que acessar a rede.

Pode ser implementado mais tarde um protocolo adaptável, que com certeza melhorará o desempenho das aplicações paralelas executadas no ambiente MMG.

4.5 - Operações de Sincronização

Para que seja garantido que apenas um processo consiga permissão para atualização de uma variável compartilhada tem-se duas primitivas de sincronização que são chamadas pelo gerente de memória compartilhada que pede um serviço para o gerente de sincronização.

As primitivas definidas pelo ambiente MMG para controle da exclusão-mútua são: **mmg_lock(S)** e **mmg_unlock(S)**.

4.6 - Modelo de Programação

A programação no ambiente MMG é realizada utilizando o paradigma de memória compartilhada distribuída. A sincronização é mantida pelo gerente de memória (GMD) através de semáforos.

As principais características deste ambiente são a utilização de primitivas simples e já conhecidas de memória compartilhada (IPC-Unix) e a sincronização através de semáforos. A comunicação entre os processos servidores de memória compartilhada é realizada através de *sockets* e totalmente transparente às aplicações.

A escolha por este modelo se deve a facilidade de programação e portabilidade de programas que foram escritos utilizando IPC. A principal preocupação do programador será a de alterar as chamadas de IPC, para as novas chamadas.

4.7 - Definição das Primitivas do Sistema

O ambiente MMG é formado por dois conjuntos de primitivas: as primitivas de usuário e as primitivas do sistema.

As primitivas de usuário são aquelas utilizadas pelo programador ao invés de utilizar as primitivas originais do IPC, para fazer abstração da memória distribuída.

As primitivas do sistema são chamadas pelas primitivas de usuário para gerenciarem a localização, criação, coerência e sincronização dos dados compartilhados.

É importante lembrar que as primitivas do ambiente são implementadas a nível do sistema operacional Linux.

4.7.1 - Primitivas de Usuário

A seguir são apresentadas as primitivas utilizadas pelo programador:

mmg_init( )

Esta primitiva inicializa o ambiente de memória compartilhada distribuída no nodo local. Faz com que o gerente de memória compartilhada fique pronto para receber requisições.

mmg_exit( )

Esta primitiva finaliza o ambiente de memória compartilhada distribuída no nodo local caso todos os dados compartilhados sob sua responsabilidade já tiverem sido liberados por todos os processos que os utilizaram.

shmget_dsm(const char *key, size_t *size, int shmflg)

Esta primitiva, como a primitiva original do IPC, retorna um *inteiro* correspondente ao identificador (*id*) da variável cujo *nome* é passado no primeiro parâmetro.

shmat_dsm(int id, void *shmaddr, int shmflg)

Retorna um ponteiro para a área identificada pelo *id* recebido pela chamada *shmget_dsm*.

shmdt_dsm(void *shmaddr)

Libera a variável compartilhada que foi anexada ao espaço de endereçamento do processo através da chamada *shmat_dsm*.

shmctl_dsm(int id, int cmd)

Esta primitiva é utilizada para eliminar a variável depois que todos os processos que fazem uso dela pedirem a sua eliminação.

A Figura 4.7.1 apresenta as primitivas a nível de usuário:

Mmg_init( )

Mmg_exit( )

Shmget_dsm(const char *key, size_t *size, int shmflg)

Shmat_dsm(int id, void *shmaddr, int shmflg)

Shmdt_dsm(void *shmaddr)

Shmctl_dsm(int id, int cmd)

Figura 4.7.1 - Primitivas do ambiente MMG a nível de usuário

A seguir é apresentado um exemplo de um programa utilizando as primitivas do ambiente MMG (Figura 4.7.2).

```
#include <sys/types.h>
#include <sys/shm.h>

int shmget_dsm(const char *key, size_t *size, int shmflg);

void *shmat_dsm(int id, void *shmaddr, int shmflg);

int shmdt_dsm(void *shmaddr);

int shmctl_dsm(int id, int cmd);

main()
{
 int id;
 size_t size;

 size = 1024;

 id = shmget_dsm("marcia", &size, 0);

 printf("id: %d\nsize: %d\n", id, size);

 return 0;
}
```

Figura 4.7.2 – Exemplo de um programa que executa no ambiente MMG.

4.7.2 Primitivas de Sistema

A seguir são apresentadas as primitivas a nível de sistema:

mmg_shmget_dsm (const char *key, size_t *size, int shmflg, pid)

Esta primitiva é chamada pela primitiva de usuário *shmget_dsm* caso a variável não exista localmente, para que localize dado compartilhado e o deixe disponível na memória local. Esta primitiva, bem como as seguintes estão detalhadas na seção 5.1, onde está descrita a implementação do gerente de memória compartilhada.

mmg_shmat_dsm (int id, void *shmaddr, int shmflg)

Esta primitiva é chamada pela primitiva de usuário *shmat_dsm*, para que realize as operações de coerência e sincronização correspondentes ao tipo de acesso requerido por *shmflg*.

mmg_shmdt_dsm (void *shmaddr)

Esta primitiva é chamada pela primitiva de usuário *shmdt_dsm*, que realiza as operações de coerência necessárias à liberação da variável, correspondente ao tipo de acesso requerido.

mmg_shmctl_dsm(int id, int cmd)

Esta primitiva realiza a liberação do segmento usado pelo dado compartilhado, depois de verificar se não existe mais nenhum processo utilizando este

segmento.

A Figura 4.7.3 apresenta as primitivas a nível de sistema:

Mmg_shmget_dsm(const char *key, size_t *size, int shmflg)

Mmg_shmat_dsm(int id, void *shmaddr, int shmflg)

Mmg_shmdt_dsm(void *shmaddr)

Mmg_shmctl_dsm(int id, int cmd)

Figura 4.7.3 - Primitivas do ambiente MMG a nível de sistema.

CAPÍTULO V

IMPLEMENTAÇÃO DO AMBIENTE MMG

Um protótipo do ambiente MMG está parcialmente implementado na Universidade do Sul de Santa Catarina, no laboratório de projetos do Curso de Ciência da Computação, em microcomputadores Pentium, interconectados através de uma rede com um *switch* de 100 Mb.

O ambiente é parte integrante do *kernel* do sistema operacional Linux. Portanto, é um ambiente homogêneo, no que diz respeito a sistema operacional. É um ambiente simples, onde a implementação ficou baseada em mudanças nas primitivas do IPC para memória compartilhada, utilizando a mesma estrutura de dados e desenvolvendo-se mais algumas primitivas para controle de sincronização e coerência distribuídas.

5.1 - Arquitetura do Ambiente

O ambiente MMG (Figura 5.1) é formado por um gerente de memória compartilhada distribuída que possui várias funções como identificar e localizar os

nodos do *cluster* e realizar as operações de sincronização distribuída.

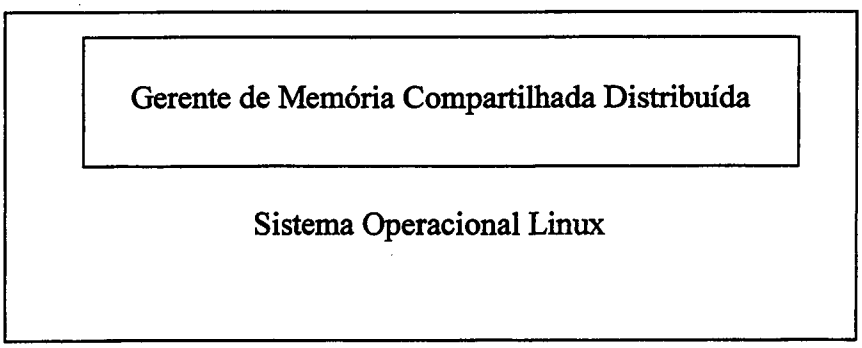

Figura 5.1 Visão geral do ambiente MMG para cada nodo.

Cada nodo do sistema possui um gerente de memória compartilhada distribuída (GMD) responsável pelas operações de sincronização e coerência dos dados compartilhados. A comunicação entre os nodos, para atualização das variáveis compartilhadas é realizada pelos GMD, utilizando *sockets*. A arquitetura básica do ambiente MMG é mostrada abaixo pela Figura 5.2.

O servidor GMD é replicado em todos os nodos para melhorar o desempenho do ambiente, o que torna o ambiente mais complexo para ser implemetado.

A seguir são descritas as funções do gerente de memória compartilhada distribuída do ambiente MMG e o seu funcionamento.

5.1.1 - Gerente de Memória Compartilhada Distribuída

O gerente possui uma tabela que contem os dados compartilhados do nodo

local, com o nome do dado, o identificador (*id*), o estado do dado (livre ou alocado para escrita), dono, variável de sincronização associada.

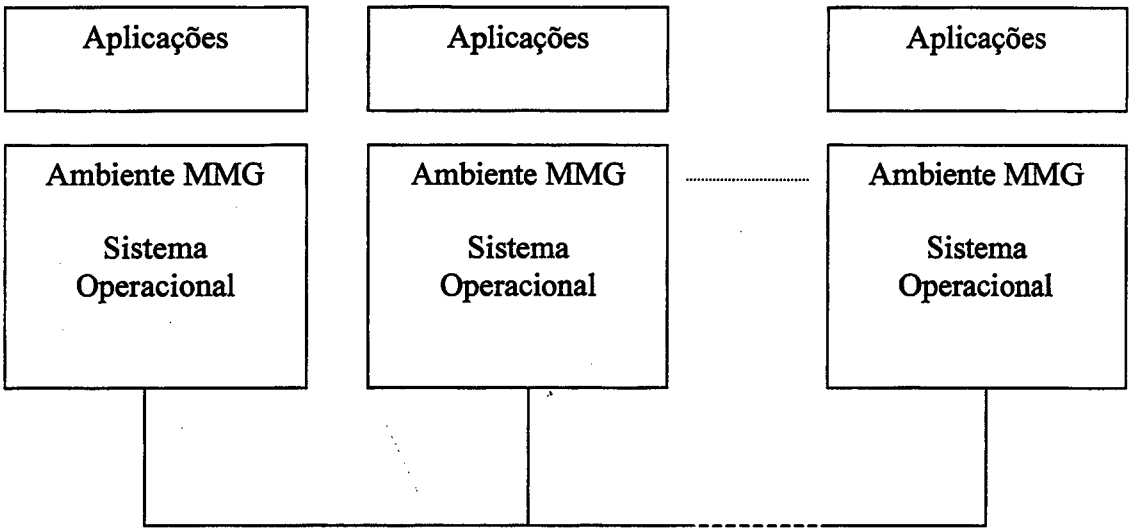

Figura 5.2 – Arquitetura básica do ambiente MMG sobre uma rede de estações de trabalho.

Cada processo ao deixar de usar o dado, pede a sua eliminação, que só é executada se não existir mais nenhum processo utilizando o dado. Para fazer este controle, o gerente de memória possui uma lista de processos que estão utilizando cada dado compartilhado.

O processo quando não quer mais utilizar um dado compartilhado faz uma chamada *mmg_shmctl_dsm* pedindo a eliminação da variável, o gerente então elimina o processo de sua lista. Quando a lista ficar vazia o gerente então elimina o dado. Dessa forma, uma área de memória compartilhada pode ser persistente.

Quando um gerente deseja obter informações sobre um determinado dado,

envia mensagens para todos os nodos e o dono (um dos gerentes GMD) do dado retorna a resposta diretamente para o gerente requisitante.

O servidor gerente de memória também possui uma tabela com o endereço dos nodos que compõem o *cluster*, carregada quando o ambiente é inicializado pela primitiva *mmg_init( )*. Esta tabela é responsável pelo armazenamento das informações relativas a todos os GMD que compõe o ambiente, como a sua localização e o *socket* de comunicação no qual aguarda uma requisição e, também cria a estrutura da tabela de memória compartilhada.

Outra funcionalidade do gerente de memória distribuída (GMD) é manter a sincronização das requisições dos demais gerentes do *cluster*. O GMD realiza atividades de criação e inicialização de semáforos e variáveis de bloqueio, bem como todas as operações sobre os mesmos, bloqueando e liberando os processos.

5.1.2 - Primitivas do Gerente de Memória Compartilhada Distribuída

A seguir são apresentadas as primitivas do gerente de memória compartilhada distribuída:

mmg_init( )

Esta primitiva inicializa o ambiente de memória compartilhada distribuída em todos os nodos do sistema. Faz com que o gerente de memória compartilhada fique pronto para receber requisições.

A inicialização do ambiente é feita em cada nodo do *cluster* pelos

as operações correspondentes ao tipo de acesso requisitado. Quando o programa de usuário faz a chamada *shmat_dsm*, essa primitiva é chamada.

A primeira verificação é se o gerente local é o dono da cópia. Se for o dono, verifica se a cópia está livre, se estiver livre continua o processo, caso contrário entra na fila de espera onde permanece até ser desbloqueado. Então o estado da variável é alterado para indicar que está sendo utilizada (leitura ou escrita) e o IPC é chamado através da primitiva *shmat*( ).

Caso o dono seja um gerente remoto, é realizada uma requisição ao dono, fornecendo o tipo de acesso a ser realizado (leitura ou escrita). O gerente dono analisa o pedido e só retorna a requisição quando a variável estiver disponível para o requisitante. A primitiva *shmat* é então chamada.

O gerente remoto faz o mesmo tratamento que o gerente local quando aquele é o dono da variável.

Campo	Tipo
Nome da variável	Char
Identificador (*id*)	Int
Estado	Int (1 – livre, 2 – alocado)
Dono	Int
Variável de sincronização	Int

Tabela 5.1.1 – Estrutura da tabela de variáveis compartilhadas do GMD

identificador (*id*).

Caso a cópia não exista localmente, é verificado nos gerentes remotos, através de envio de mensagens a todos os nodos. O gerente dono do dado retorna as informações referentes a ela como valor, tamanho do segmento e variável de sincronização associada. O gerente requisitante então pega os dados e chama a primitiva original do IPC de memória compartilhada, passando o tamanho do segmento que retornou da requisição e os demais parâmetros que foram informados na primitiva *shmget_dsm*, e o dono do dado fica sendo o gerente criador.

Se o dado não tiver sido criado em nenhum nodo do sistema, é chamada a primitiva original do IPC com os parâmetros informados pelo programa na primitiva *shmget_dsm*.

Após o retorno da chamada da primitiva original, que retorna o *id* correspondente, é feita a entrada na tabela de variáveis compartilhadas do gerente local.

Note-se que a criação da variável compartilhada é sempre feita pela primitiva original do IPC.

A figura 5.4 a seguir apresenta um fluxo de execução da primitiva mmg_shmget_dsm.

A tabela que o GMD mantém com os dados compartilhados possui a seguinte estrutura (Tabela 5.1.1).

mmg_shmat_dsm (int id, void *shmaddr, int shmflg)

Esta primitiva é responsável por manter a coerência de memória. Executa

as operações correspondentes ao tipo de acesso requisitado. Quando o programa de usuário faz a chamada *shmat_dsm*, essa primitiva é chamada.

A primeira verificação é se o gerente local é o dono da cópia. Se for o dono, verifica se a cópia está livre, se estiver livre continua o processo, caso contrário entra na fila de espera onde permanece até ser desbloqueado. Então o estado da variável é alterado para indicar que está sendo utilizada (leitura ou escrita) e o IPC é chamado através da primitiva *shmat*( ).

Caso o dono seja um gerente remoto, é realizada uma requisição ao dono, fornecendo o tipo de acesso a ser realizado (leitura ou escrita). O gerente dono analisa o pedido e só retorna a requisição quando a variável estiver disponível para o requisitante. A primitiva *shmat* é então chamada.

O gerente remoto faz o mesmo tratamento que o gerente local quando aquele é o dono da variável.

Campo	Tipo
Nome da variável	Char
Identificador (*id*)	Int
Estado	Int (1 – livre, 2 – alocado)
Dono	Int
Variável de sincronização	Int

Tabela 5.1.1 – Estrutura da tabela de variáveis compartilhadas do GMD

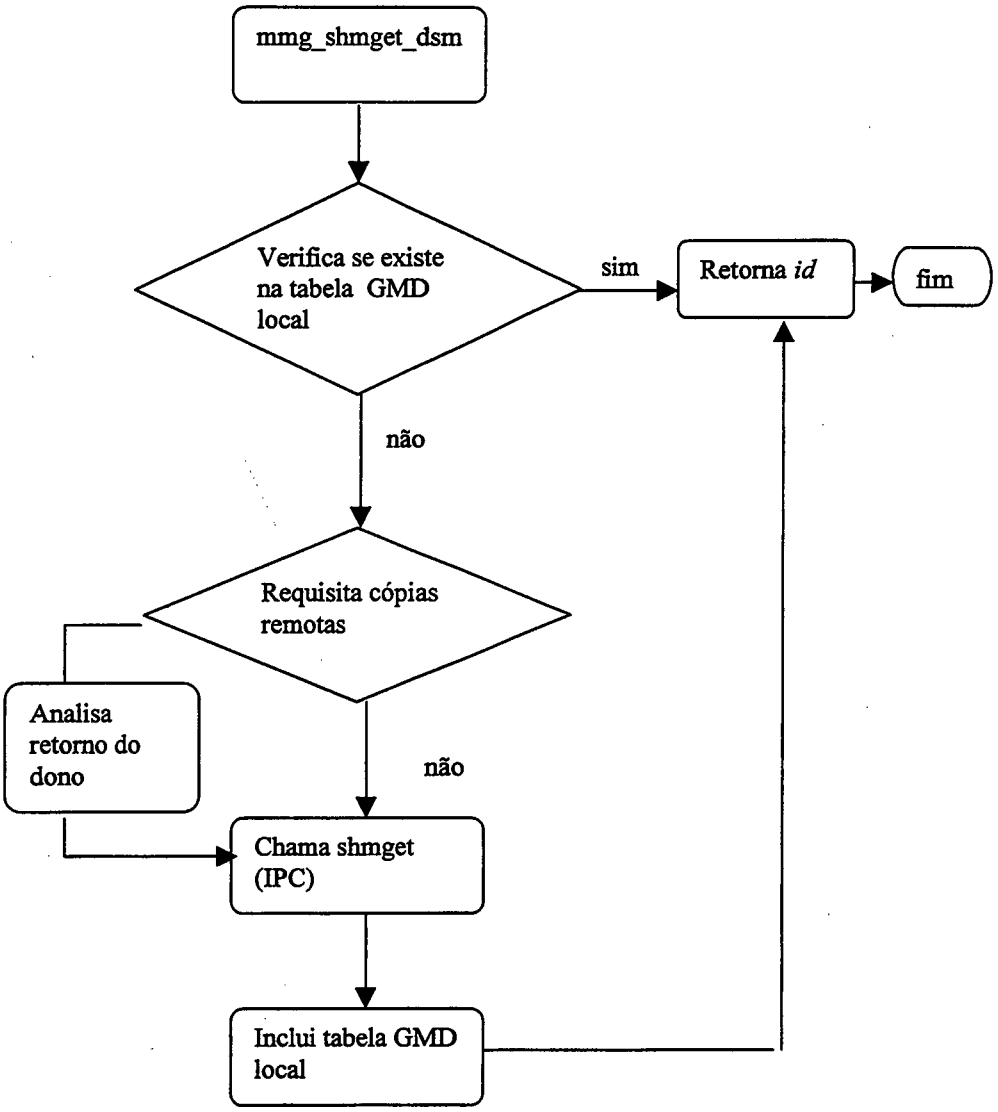

Figura 5.4 – Fluxo de execução da primitiva mmg_shmget_dsm.

A seguir na figura 5.5 é apresentado o fluxo de execução da primitiva mmg_shmat_dsm.

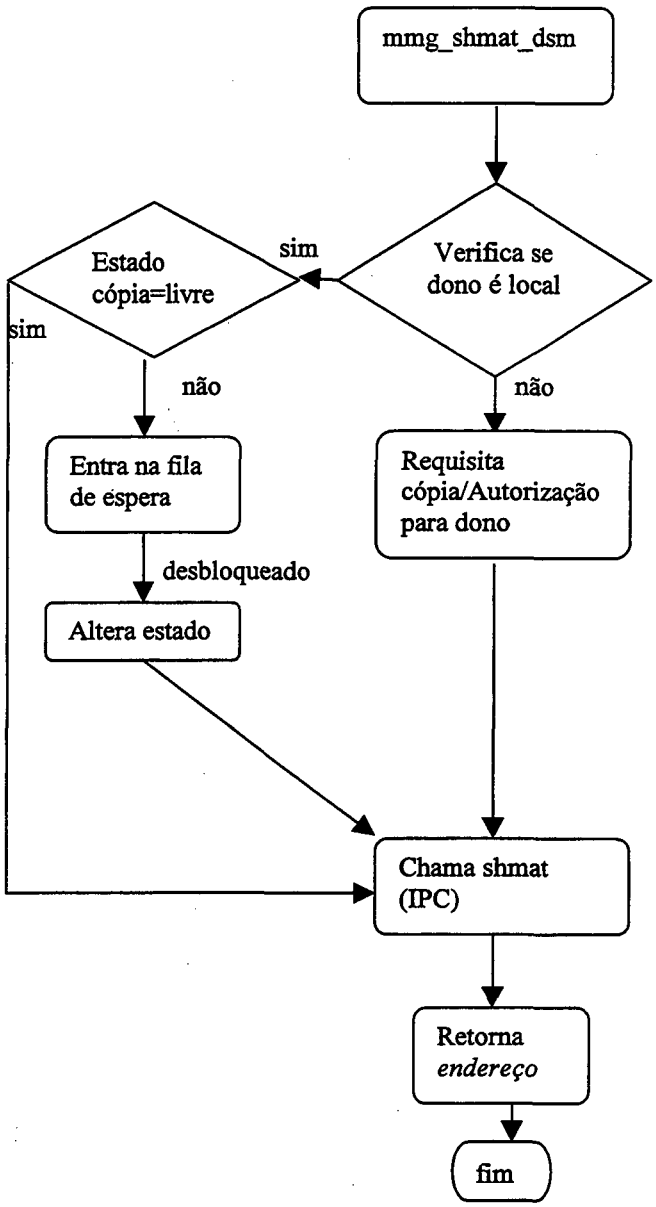

Figura 5.5 – Fluxo de execução da primitiva mmg_shmat_dsm.

mmg_shmdt_dsm (void *shmaddr)

Esta primitiva é acionada quando um processo cliente local executa um *shmdt*( ) ou, quando recebe uma mensagem de um servidor de memória de outro nodo, com atualização de cópia de uma variável compartilhada. A figura 5.6 mostra o fluxo de execução desta primitiva.

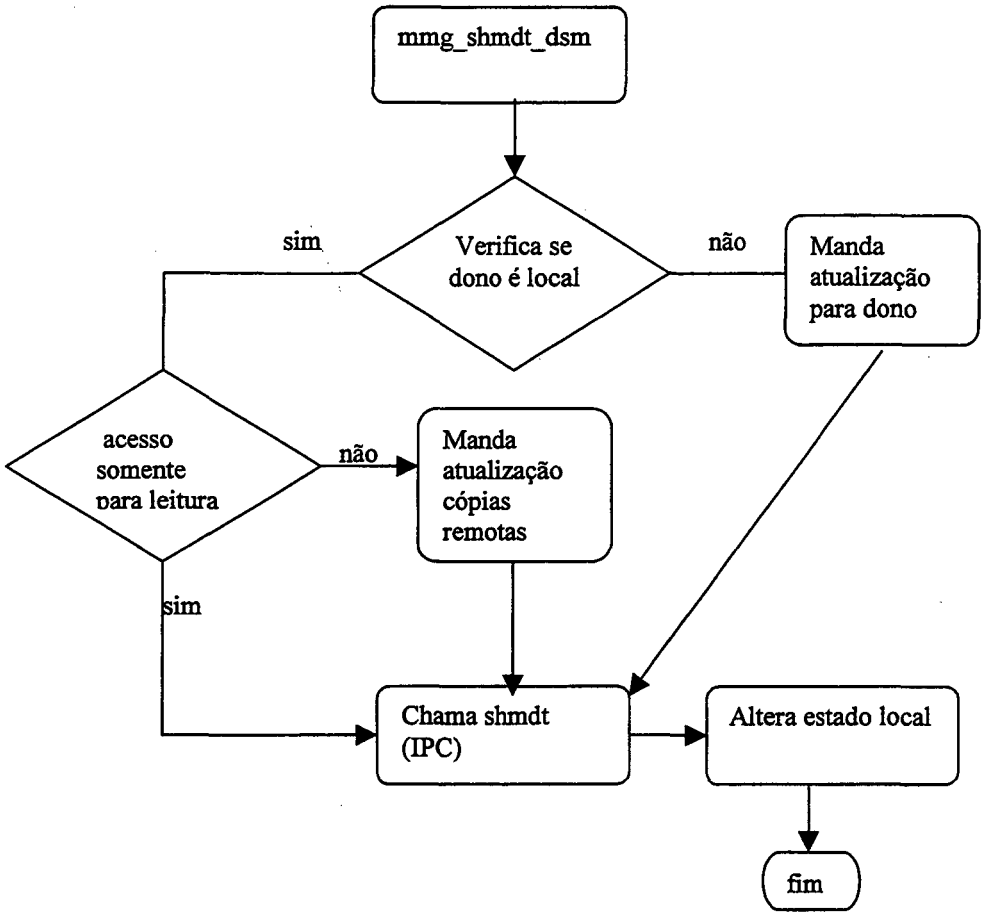

Figura 5.6 – Fluxo de execução da primitiva mmg_shmdt_dsm.

CAPÍTULO VI

CONCLUSÃO

Este trabalho apresentou a definição de um ambiente de programação paralela, para arquiteturas que não possuem memória compartilhada fisicamente. Este modelo oferece maior facilidade de programação e portabilidade de aplicações já escritas utilizando primitivas de memória compartilhada da API do UNIX referentes a comunicação entre processos.

O modelo foi desenvolvido com uma abordagem simples que não demanda uma estrutura complexa. O sistema utiliza chamadas de sistema do UNIX para comunicação entre processos baseada em memória compartilhada, estendendo-as para um ambiente distribuído.

Para validar o modelo, foi desenvolvido parcialmente um protótipo para ser executado em apenas um nodo. Os demais nodos são simulados pela criação de processos gerentes de memória (GMD), onde cada estrutura de gerente simula um nodo. O gerentes de memória (GMD) comunicam-se através de *sockets*.

A coerência de memória implementada no protótipo foi a coerência de entrada, por se tratar da coerência mais relaxada, que possibilita um melhor desempenho.

Por fim, no trabalho apresentado, pode-se destacar como principais contribuições: a concepção de um ambiente de programação paralela que facilita a implementação dos programas, principalmente, para os programadores que já utilizam chamadas de sistema IPC que implementam memória compartilhada; uma abordagem nova de ambiente MCD, pois o modelo foi desenvolvido para ser implementado como uma funcionalidade do sistema operacional.

6.1 - Trabalhos Futuros

Um importante trabalho a ser realizado é a incorporação do ambiente no mecanismo de IPC de memória compartilhada. Já que o protótipo foi implementado fazendo chamadas de IPC. Este trabalho é fundamental para que se realize a avaliação de desempenho e comparações com outros sistemas MCD.

Para tornar o ambiente mais dinâmico é interessante que sejam desenvolvidos outras semânticas de coerência como sequencial e de processador. Onde o programador possa optar entre uma delas, dependendo do tipo de aplicação a ser executada no ambiente.

No protótipo foi implementado o protocolo de escrita-atualização, que foi o suficiente para a validação do modelo. Entretanto, a implementação de um protocolo adaptável pode melhor o desempenho das aplicações.

CAPÍTULO VII

REFERÊNCIAS BIBLIOGRÁFICAS

[DEL91] DELP, G. et. al. **Memory as a Network Abstraction.** IEEE Network, July 1991, p. 34-41.

[LI89] LI, K., and P. HUDAK, **Memory Coherence in Shared Virtual Memory Systems,** *ACM Transactions on Computer Systems*, v. 7, n°. 4, 1989.

[COU95] COULOURIS, George, DOLLIMORE, Jean and KINDBERG, Tim. **Distributed Systems - Concepts and Design.** Great Britain, Addison-Wesley, 1995, 644 p.

[HWA93] HWANG, Kai. **Advanced Computer Architecture:** Parallelism, Scalability, Programmability. New York: MacGraw-Hill, 1993, 771 p.

[JAM94] JAMES, D. V. **The Scalable Coherent Interface: Scaling to High-Performance Systems**. Compcon 94, IEEE Computer Society Press, Los Alamitos, 1994, p. 64-71.

[LO 94] LO, V. **Operating Systems Enhancements for Distributed Shared Memory**. Advances in Computer. Academic Press, San Diego, v. 39, p. 191-237, 1994.

[TAN95] TANENBAUM, A. S. **Distributed Operating Systems**. Prentice Hall, 1995.

[LEN92] LENOSKI, D. et. al. **The Stanford DASH Multiprocessor. Computer**, v. 25, n° 3, March 1992, p. 63-79.

[LUC95] LUCCI, S. et. al. **Reflective-Memory Multiprocessors.** Proc. 28th IEEE/ACM Hawaii Int'l Conf. Systems Siences. IEEE Computer Society Press, Los Alamitos, 1995, p. 85-94.

[MAP90] MAPLES, C. & WITTIE, L. **Merlin: A Superglue for Multicomputer Systems.** Compcon 90. IEEE Computer Society Press, Los Alamitos, 1990, p. 73-81.

[MIL99] MILUINOVIC, V. & STENSTRÖN, P. **Scanning the Issue: Special Issue on Distributed Shared Memory Systems**. Proceedings

of the IEEE. March 1999, v. 87, n° 3.

[FRA93] FRANK, S. et al. **The KSR1: Bridgring the Gap Between Shared Memory and MMPs**. Compcon 93, IEEE Computer Society Press, Los Alamitos, 1993, p. 285-294.

[LDP98] **The Linux Documentation Project**. Disponível via WWW em http://sunsite.unc.edu/LDP/HOWTO/ agosto/1998

[BEC98b] BECKER, Donald. **Beowulf Software**. Disponível via WWW em http://cesdis.gsfc.nasa.gov/linux/beowulf/software/software.html agosto/1998

[RID97] RIDGE, Daniel; BECKER, Donald; MERKEY, Phillip; STERLING, Thomas. **Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs**. Proceedings, IEEE Aerospace, 1997. Disponível via WWW em http://cesdis.gsfc.nasa.gov/beowulf/papers/AA97/aa97.ps, dezembro/1998

[PRO96] PROTIC, J. et al. **Distributed Shared Memory: Concepts and Systems**. IEEE Parallel and Distributed Technology. Summer 1996. v. 4, n°. 2.

[ZWA94] ZWAENEPOEL, Willy et alli.. **TreadMarks: Shared Memory Computing on Standard Workstations and Operation Systems.** In Proceedings of the 1994 Winter Usenix Conference, p. 115-131, 1994.

[ZWA96] ZWAENEPOEL, WILLY et alli.. **TreadMarks: Shared Memory Computing on Standard Workstations.** IEEE, 1996, p. 18-28.