MIGRAÇÃO DE UM SISTEMA DE GRANDE PORTE BASEADO EM MEMÓRIA-COMPARTILHADA PARA UM AMBIENTE DE MEMÓRIA-COMPARTILHADA DISTRIBUÍDA

Tatiana Cavalcanti Fernandes

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

Aprovada por:

Prof. Inês de Castro Dutra, Ph.D.

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Maria Clicia Stelling de Castro, D.Sc.

Prof. Alexandre Sztajnberg, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 2004

FERNANDES, TATIANA CAVALCANTI

Migração de um sistema de grande porte baseado em memória-compartilhada para um ambiente de memória-compartilhada distribuída [Rio de Janeiro] 2004

XII, 109 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2004)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Sistemas de memória centralizada

2 - Sistemas de memória-compartilhada distribuída

3 - Paralelismo em Programação em Lógica

4 - Paralelismo OU

5 - Aurora

I. COPPE/UFRJ II. Título (série)

A minha querida mãe Celia Maria Cavalcanti Fernandes

Agradecimentos

A Deus, pelo presente da Vida. Por estar ao meu lado em todos os momentos. Por me sustentar, proteger e guiar. Por ser meu refúgio e minha paz. Por ser Aquele que, nas tribulações, me levanta e me orienta; que renova minha esperança e minha fé; enfim, que me faz forte. E quando tudo parece perdido, é Ele quem sempre me diz: "Não desista, siga em frente porque você vai conseguir". E eu consigo.

A minha querida mãe, a quem devo tudo que sou. Por todas as dificuldades que passamos juntas, estar aqui é muito bom. Faz a vida valer a pena. Obrigada por toda a dedicação na nossa formação (minha e do meu irmão), por ter nos mostrado o que é ter honra, caráter e bondade. Por ter nos mostrado o amor. Ainda me lembro dos tempos de infância, do seu apoio nos meus primeiros dias de aula... eu tinha tanto medo. E a saída da escola? Era só eu vê-la chegar, com a alegria de sempre estampada em seu rosto, que meu cansaço desaparecia. Isso é uma das coisas de que mais me lembro. E o seu exemplo, sempre me motivou a seguir em frente, mesmo diante das maiores adversidades. E, por isso, também, sempre fui tão dedicada às coisas que fiz. Eu precisava recompensá-la de alguma forma, apesar de você nunca ter me exigido nada.

Ao meu esposo, João Mauricio, por todo o amor, carinho, companheirismo e apoio de sempre. Pela revisão do texto e debates técnicos que contribuíram muito para esta tese. Por toda ajuda e dedicação desde os tempos da graduação. Você é meu anjinho da guarda de todos os dias; aquele que me protege e faz minha vida muito feliz.

Ao meu irmão André Leonardo e minha avó Nadina pelo amor e suporte tão importantes em minha vida.

A minha orientadora, Profª Inês de Castro Dutra, pelo crescimento profissional adquirido. Obrigada pelo carinho e parabéns pelo ser humano especial que você é.

Ao meu professor e orientador da graduação, Prof. Alexandre Sztajnberg, por ter me incentivado a ingressar no mestrado e, principalmente, pela motivação e carinho

de sempre. Por ser um exemplo de profissional, competente e dedicado, e de ser humano, pelo seu caráter marcante. Obrigada por acreditar em mim.

Aos professores que compuseram a banca examinadora: Claudio Luis de Amorim, Maria Clicia Stelling de Castro e Alexandre Sztajnberg.

Aos professores da COPPE-Sistemas, especialmente aos da linha de Arquitetura e Sistemas Operacionais (ASO).

Ao eterno Ayrton Senna, meu maior exemplo de determinação, dedicação, fé e garra. Ele me ensinou que é possível superar limites. Seus ensinamentos vão estar sempre comigo.

"Seja você quem for, seja qual for a posição social que você tenha na vida, a mais alta ou a mais baixa, tenha sempre como meta muita força, muita determinação e sempre faça tudo com muito amor e com muita fé em Deus, que um dia você chega lá. De alguma maneira, você chega lá."(Ayrton Senna)

Ao Pe. Marcelo Rossi, pelas palavras diárias de fé através do programa Momento de Fé. Obrigada por se permitir ser um instrumento tão especial de Deus, e me ensinar a amar e confiar em Jesus em todos os momentos.

A minha amiga, Carla dos Santos Santana, pela revisão no texto e amizade sincera.

Aos meus amigos do Colégio Pedro II e UERJ. Aos meus amigos da COPPE pelo companheirismo e brincadeiras que tornaram o mestrado mais divertido. Ao Anderson Faustino, pelas explicações importantes sobre o funcionamento do TreadMarks.

E por último, mas não menos importante, ao CNPq pelo financiamento dessa pesquisa.

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

MIGRAÇÃO DE UM SISTEMA DE GRANDE PORTE BASEADO EM MEMÓRIA-COMPARTILHADA PARA UM AMBIENTE DE MEMÓRIA-COMPARTILHADA DISTRIBUÍDA

Tatiana Cavalcanti Fernandes

Março/2004

Orientadora: Inês de Castro Dutra

Programa: Engenharia de Sistemas e Computação

O desenvolvimento dos microprocessadores e das redes de alta velocidade contribuiu para o estudo de uma nova arquitetura de computadores, que visava dar continuidade aos avanços obtidos em desempenho, pela interligação de múltiplos processadores. Assim surgiu a computação distribuída; nela o trabalho exigido por uma aplicação pode ser dividido em diversas máquinas, aproveitando os recursos disponíveis na rede.

Nesse contexto, os sistemas de memória-compartilhada distribuída (DSM - *Distributed Shared-Memory*) têm o objetivo de implementar, em nível lógico, a abstração do modelo de memória-compartilhada, embora fisicamente a mesma esteja distribuída. Através deles, aplicações projetadas para arquiteturas de memória centralizada podem ser executadas em plataformas de memória distribuída, mantendo, respectivamente, a facilidade de programação e a escalabilidade inerente de tais arquiteturas.

Diversos estudos foram feitos sobre sistemas DSM, relacionados à adaptação de aplicações projetadas para arquiteturas de memória centralizada a sistemas DSM, de forma que as mesmas pudessem ser executadas em plataformas de memória distribuída. Mas, os resultados encontrados restringem-se à análise de aplicações de pequeno e médio porte no que diz respeito a softwares DSM.

Este trabalho vem analisar a complexidade de se portar um sistema de grande porte para um software DSM. Objetiva-se compor uma referência sobre tal assunto e resultados quanto a: medidas de desempenho obtidas e viabilidade de utilização em conjunto de tais sistemas. Para tanto, utiliza-se o software DSM TreadMarks e o sistema Aurora, uma implementação protótipo de paralelismo OU da linguagem Prolog para multiprocessadores de memória-compartilhada.

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

MIGRATION OF A LARGE-SCALE SHARED-MEMORY SYSTEM TO A DISTRIBUTED SHARED-MEMORY ENVIRONMENT

Tatiana Cavalcanti Fernandes

March/2004

Advisor: Inês de Castro Dutra

Department: Computing and Systems Engineering

The development of microprocessors and high-speed networks contributed to the research of a new computer architecture, which aimed to carry on the advances in performance gained until then, by joining many processors together. That was the beginning of distributed computing. Here, the workload of an application may be distributed among multiple machines, using the available resources in networks.

In this context, distributed shared-memory systems (DSM) intend to implement, in a logical level, an abstraction of the shared-memory model, although the memory is physically distributed. Through those systems, applications constructed for memory architectures may be executed in distributed memory platforms. Besides that, they are able to maintain the advantages provided for such architectures - the easiness of programming the shared-memory and the scalability obtained from distributed-memory architectures.

Many studies about DSM systems were made, aimed to adapt applications initially designed for centralized memory architectures to DSM systems, so that those applications could be executed in distributed memory platforms. However, the results obtained are restricted to the analysis of small and medium-size applications concerning software DSM.

The aim of this work is to analyze the complexity obtained to port a large-sized system to a software DSM, so that we can have a reference about this subject and results concerning to: obtained performance measures and the viability of by-and-large utilization of such systems. In order to do that, the software DSM TreadMarks and the Aurora system - an OR-parallelism prototype implementation of Prolog language for shared-memory multiprocessors - are used.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Introdução

O desenvolvimento constante e ininterrupto de novas tecnologias, ao longo das últimas décadas, tem promovido mudanças significativas nos diversos setores da computação, introduzindo conceitos, ultrapassando antigos limites e propondo novos desafios. Assim surgiu a computação distribuída, uma evolução da computação seqüencial que visa fornecer alto poder de computação e baixos custos, através da utilização de múltiplos recursos. Seu estudo foi favorecido pelo surgimento dos microprocessadores e das primeiras redes locais de alta velocidade.

A tecnologia dos microprocessadores deu maior capacidade de processamento às máquinas, mas não resolveu os diversos problemas que impunham limites à computação seqüencial. No mesmo período, as primeiras redes de alta velocidade foram desenvolvidas, com o objetivo de conectar diversas máquinas e permitir a troca de pequenas quantidades de informações. Desse modo, despontaram as redes de computadores, que passaram a constituir o cenário predominante nos principais centros de pesquisa e grandes corporações.

Todos esses fatores contribuíram para o estudo de uma nova arquitetura de computadores. Esta visava dar continuidade aos avanços obtidos em desempenho até aquele momento, pela interligação de múltiplos processadores por meio de redes de alta velocidade. Assim, os sistemas distribuídos e os sistemas paralelos despontaram como fontes promissoras de avanço na área computacional, tornando-se focos de estudos até os dias de hoje. Neles, o trabalho exigido por uma aplicação pode ser dividido em diversas máquinas, aproveitando os recursos disponíveis na rede e acelerando o processo de resolução dos problemas.

A disseminação das redes de computadores, apesar de todos os benefícios,

trouxe também alguns problemas a serem resolvidos: (1) existem diversas aplicações relevantes para a ciência que, cada vez mais, tem seu uso restrito pela arquitetura, pelo fato de terem sido projetadas para plataformas de memória-compartilhada; (2) desenvolvendo-se sistemas para redes de memória distribuída, perde-se a facilidade oferecida pela programação em memória-compartilhada; nela, o programador não precisa se preocupar em distribuir os dados pelos processos.

Na tentativa de solucionar esses problemas, surgiram propostas para o desenvolvimento de sistemas de memória-compartilhada distribuída (DSM - *Distributed Shared-Memory*), cujo modelo de comunicação implementa em nível lógico a abstração do modelo de memória-compartilhada, embora fisicamente a mesma esteja distribuída. Essa abstração mantém a facilidade de programação e portabilidade das arquiteturas de memória centralizada. Além disso, torna transparente ao programador o mecanismo de troca de mensagens existente nas arquiteturas de memória-distribuída, mantendo a escalabilidade e poder de computação característico de tais sistemas.

1.2 Motivação

Nos últimos anos, a ubiqüidade das redes de computadores possibilitou o estudo de novas alternativas para os problemas enfrentados pelo processamento seqüencial. Como resultado, a computação distribuída firmou-se como uma fonte de obtenção de melhores desempenhos e tempos de resposta para os usuários.

Nesse contexto, o desenvolvimento de softwares DSM, para dar continuidade à usabilidade dos sistemas de memória-compartilhada, abriu novos caminhos e perspectivas futuras, tornando-se alvo de diversos estudos até os dias atuais.

O cenário exposto aliado à avaliação da quantidade de sistemas de memória-compartilhada que, gradualmente, têm sua aplicabilidade restrita pela arquitetura, motivou o estudo do tema desta dissertação.

1.3 Objetivos

Diversos estudos foram realizados sobre software DSM. Esse trabalho, entretanto, tem o objetivo de avaliar a complexidade de se adaptar uma aplicação de grande porte, escrita originalmente para memória-compartilhada, a arquiteturas de memória distribuída. A migração foi realizada em ambiente Linux, onde se utilizou

o software DSM TreadMarks, uma biblioteca em nível de usuário de sistemas Unix, que pode ser combinada com programas escritos em C, C++ e Fortran.

Para realizar essa adaptação, optou-se pelo sistema Aurora, uma implementação protótipo de paralelismo OU da linguagem Prolog para multiprocessadores de memória centralizada. A escolha baseou-se na relevância desse sistema para a área de Inteligência Artificial como: (a) ferramenta de pesquisa de sistemas de programação lógica em paralelo e (b) sistema de demonstração para a execução de aplicações paralelas de grande porte.

1.4 Contribuições da Autora

As pesquisas desenvolvidas, sobre sistemas DSM, concentram-se em avaliar características dos softwares, como: ferramentas disponíveis, complexidade envolvida na migração de sistemas de pequeno e médio porte, assim como o desempenho obtido. Entretanto, não se tem dados sobre a complexidade ou desempenho obtidos no processo de adaptação de sistemas de grande porte para um software DSM.

Desse modo, este trabalho tem a contribuição de constituir uma referência de análise da complexidade e desempenho em se migrar um sistema de grande porte para um software DSM.

1.4.1 Organização do Trabalho

O presente texto está estruturado em capítulos, cujos temas concentram-se em proporcionar informações que, gradualmente, contribuam para o entendimento do trabalho de pesquisa desenvolvido.

O Capítulo 2 apresenta os conceitos de processamento paralelo e sistemas DSM. As Seções 2.1 e 2.2 fornecem uma visão geral da computação paralela para o leitor iniciante no assunto. As Seções 2.3, 2.4 e 2.5 introduzem os conceitos relevantes de memória, tais como organização e modelos de comunicação de dados, assim como os relacionados à memória-compartilhada distribuída. A Seção 2.6 apresenta o software DSM TreadMarks, suas propriedades, características e funcionamento. Finalmente, a Seção 2.7 insere algumas noções de programação em lógica e os tipos de paralelismo existentes nessa abordagem.

O Capítulo 3 disserta sobre o sistema Aurora, seu modelo de execução e sua

implementação. São estudados seus dois módulos principais, o escalonador de Bristol e a Máquina Prolog (*Engine*), assim como as interfaces providas pelos mesmos para a execução do sistema.

O Capítulo 4 relata o processo de migração do sistema Aurora para o TreadMarks, os passos executados, os problemas encontrados e as soluções propostas.

O Capítulo 5 mostra os resultados obtidos na execução do sistema, após a implementação dos passos do capítulo anterior. Faz-se uma análise dos dados fornecidos pelo TreadMarks, relacionando-os à versão original.

O Capítulo 6 encerra o trabalho expondo as conclusões e propostas para pesquisas futuras, a partir da análise dos problemas e resultados encontrados nos Capítulos 4 e 5.

Capítulo 2

Paradigmas da Computação Paralela

2.1 Visão Geral da Computação Paralela

Ao longo das últimas décadas, o advento de novas tecnologias tem provocado mudanças graduais importantes no projeto e implementação da arquitetura dos computadores, introduzindo novos conceitos e possibilitando a descoberta e o estabelecimento de novos limites a serem contornados, ou mesmo superados, para se dar continuidade ao progresso. Desse desenvolvimento surgiu a computação distribuída, que visa, principalmente, acelerar o processo de resolução dos problemas computacionais pela utilização de múltiplos recursos.

2.1.1 A Origem da Computação Paralela

Inicialmente, o modelo de computação era seqüencial, ou seja, as instruções de um programa eram executadas ordenadamente, uma após a outra, em um computador convencional, composto apenas de uma Unidade Central de Processamento. No período de 1945 a 1985, essas máquinas ocupavam muito espaço e possuíam custo elevado, o que dificultava a disseminação de seu uso mesmo nas grandes corporações e centros acadêmicos. Nesses locais, *a priori*, utilizou-se um número reduzido de computadores que operavam de modo isolado, por não haver uma maneira segura de conectá-los. Posteriormente, terminais foram interligados a essas máquinas, as quais passaram a repartir o tempo de utilização de seu processador entre os mesmos, introduzindo o conceito de tempo compartilhado e multiprogramação. Havia ainda os minicomputadores, cuja maior vantagem incidia em seus tamanhos reduzidos comparados às grandes máquinas da época. Entretanto, apresentavam menor desempenho e, apesar dos menores custos, eram ainda bastante onerosos.

O ápice do processamento seqüencial foi atingido no período de 1985 a 1995,

quando os microprocessadores foram desenvolvidos, com um nível de integração muito alto. Estes tornaram-se a tecnologia dominante de processadores, trazendo consigo crescentes índices de desempenho e menores custos. Conseqüentemente, houve um crescimento significativo na aquisição e instalação de computadores em organizações como empresas, meios acadêmicos e centros de pesquisa, assim como em residências para uso pessoal e profissional. Todavia, a despeito de todo desenvolvimento, foram detectados alguns problemas que impuseram limites à computação seqüencial e fizeram com que novas soluções de arquitetura fossem procuradas. A seguir, são apresentados os principais motivos que caracterizaram tal quadro [52, 85]:

- **Limitações inerentes das máquinas von Neumann tradicionais:** o desempenho de um computador seqüencial é dependente da velocidade com que os dados são transmitidos através do hardware. E esta, por sua vez, não pode crescer indefinidamente; existe o limite da velocidade da luz, um problema enfrentado pelos projetistas atuais;

- **Limites de miniaturização:** uma alternativa para se aumentar a velocidade das máquinas é aproximar, ao máximo, todos os seus componentes. Para isso, esforços têm sido feitos no sentido de alcançar esse objetivo. Por exemplo, a tecnologia de processadores tem conseguido concentrar um número cada vez maior de transistores em um mesmo *chip*. Contudo, mesmo com componentes de nível atômico ou molecular, um limite será alcançado, quando então determinar-se-á o quão pequenos os componentes podem e devem ser; e

- **Limitações econômicas:** o custo do projeto e desenvolvimento de processadores mais velozes é bastante elevado, requer gastos excessivos de tempo e não acompanha o ritmo de exigência e necessidade dos usuários por acesso rápido às informações.

Enquanto a computação seqüencial experimentava os resultados positivos da introdução da tecnologia dos microprocessadores, e encontrava os obstáculos iniciais a serem vencidos, surgiram as primeiras redes locais de alta velocidade ou LANs (*Local Area Networks*). Estas conectaram diversas máquinas e permitiram a troca de pequenas quantidades de informações entre as mesmas. O emprego dessas duas tecnologias (microprocessadores e redes locais), juntamente com os problemas

apresentados pelo processamento seqüencial, contribuiu para o estudo de uma outra arquitetura de computadores, que visava interligar múltiplos processadores através de redes de alta velocidade. Essa arquitetura deu origem aos sistemas distribuídos, em contraste com os sistemas centralizados (sistemas paralelos e seqüenciais), que possuíam memória centralizada e um único espaço de endereçamento.

Embora muitas vezes não se faça distinção na utilização dos termos sistemas distribuídos e sistemas paralelos, existe uma pequena diferença entre os seus conceitos. Os **sistemas distribuídos** são executados em arquiteturas com dois ou mais processadores que, interligados, permitem o trabalho em conjunto de diversos usuários, geralmente realizando tarefas distintas. De modo semelhante, os **sistemas paralelos** são aplicados sobre arquiteturas com múltiplos processadores, mas o objetivo destes é obter o melhor desempenho na solução de um problema computacional. Assim, geralmente, realizam o processamento simultâneo de uma única tarefa, onde não há interação entre os usuários [86].

A partir das definições de sistemas distribuídos e paralelos, podemos generalizar o conceito de sistemas com múltiplos processadores para: **arquiteturas que interligam dois ou mais processadores para a execução, em conjunto, de tarefas independentes, ou para a execução simultânea de uma única tarefa** [61].

Nesta dissertação, ao abordar-se as arquiteturas de múltiplos processadores, é dada ênfase nos sistemas paralelos (ou computação paralela), motivada pela afinidade com a aplicação tema deste trabalho, Aurora. Esta última objetiva explorar o paralelismo existente em sistemas de programação em lógica, aumentando a velocidade na execução dos mesmos, como é visto no Capítulo 3.

2.1.2 Objetivos e Motivações da Computação Paralela

De uma maneira simplificada, a computação paralela consiste na utilização simultânea dos múltiplos recursos de uma ou várias máquinas para a resolução de um problema computacional. Normalmente, este problema é dividido em tarefas menores que são, então, distribuídas entre os processadores componentes da arquitetura, e executadas simultaneamente, potencialmente permitindo que o trabalho seja concluído em menor tempo. Os recursos utilizados para o processamento paralelo podem compreender: um computador convencional com múltiplos processadores, um número arbitrário de computadores interligados através

de uma rede de interconexão, ou mesmo uma combinação de ambas as arquiteturas anteriores [52].

Como visto anteriormente, a computação paralela é uma evolução da computação seqüencial e surgiu na década de 1960, quando os primeiros sistemas foram empregados, principalmente, em centros de pesquisa e meios acadêmicos. Seus principais objetivos eram:

- **Simular o comportamento dos sistemas existentes no mundo real**, onde há muitos eventos complexos, inter-relacionados, que ocorrem ao mesmo tempo, ainda que dentro de uma seqüência. Exemplos: linha de montagem de automóveis, processo de compra de um lanche em um *drive-thru*, a construção de um shopping, tráfego no horário de *rush* nas grandes cidades e as órbitas planetárias e galácticas [52]; e

- **Acelerar a execução de aplicações que lidam com um grande volume de cálculos** e, portanto, necessitam de um grande poder computacional, como sistemas de previsão do tempo, dispositivos mecânicos (de próteses a espaçonaves), dinâmica dos fluidos, genoma humano, atividade sísmica e reações nucleares e químicas.

Na década de 1980, com o surgimento dos microprocessadores, das LANs e dos primeiros obstáculos à computação seqüencial, teve início o fortalecimento e a disseminação das arquiteturas com múltiplos processadores. As empresas começaram a empregar o processamento paralelo, em suas aplicações comerciais, com o objetivo de melhorar o desempenho e oferecer melhores tempos de resposta aos usuários interativos. Descobriram, posteriormente, que outras vantagens poderiam ser obtidas, como aumentar a confiabilidade, a escalabilidade, a disponibilidade e o balanceamento de carga em suas aplicações, como é visto na próxima seção. Atualmente, os sistemas de processamento paralelo são a arquitetura base de aplicações comerciais, tais como: servidores de banco de dados, servidores de arquivos e servidores Web, *data mining*, sistemas de realidade virtual e gráficos avançados, ambientes de trabalho colaborativo, vídeo em rede e tecnologias multimídia, exploração de óleo, diagnósticos auxiliados por computadores na medicina e gerenciamento de corporações nacionais e multinacionais. Devido à redução de custo dessas arquiteturas e à adição, aos sistemas operacionais, de suporte

ao processamento paralelo, muitas estações de trabalho e computadores pessoais também já fazem parte de arquiteturas de múltiplos processadores [61, 52].

Entretanto, além dos objetivos expostos, esses sistemas foram estudados, desenvolvidos e aplicados motivados por diversos fatores que contribuíram para a atual tendência à descentralização. A seguir, são apresentadas as primeiras motivações para a pesquisa na área objeto de estudo deste trabalho [61, 52, 86]:

- **Economia:** o custo do projeto e desenvolvimento de processadores mais velozes é bastante elevado e a melhora obtida no desempenho é, muitas vezes, insignificante. Juntando-se um conjunto de processadores mais baratos e mais lentos, pode-se conseguir melhores resultados no desempenho a custos reduzidos. Assim, os sistemas distribuídos possuem melhor relação custo/desempenho sobre os centralizados;

- **Distribuição inerente:** muitas aplicações são inerentemente distribuídas, como um sistema de automação industrial que controla robôs e máquinas ao longo de um processo de montagem. Se cada um dos robôs e máquinas tiver seu próprio processador, e estes estiverem conectados, tem-se um sistema distribuído de automação industrial; e

- **Desempenho**[1] **:** Algumas aplicações manipulam grandes volumes de dados e, por isso, requerem máquinas com maior poder de computação. Utilizando-se múltiplos processadores, pode-se conseguir um aumento no *throughput*[2] do sistema e uma redução importante nos tempos de processamento e resposta[3] dos mesmos.

A evolução na pesquisa e no estudo dos sistemas com múltiplos processadores, inicialmente motivados pelos fatores mencionados anteriormente, conduziu à descoberta de motivações adicionais, assim como de aspectos negativos, identificados, respectivamente, como as vantagens e desvantagens de tais sistemas.

[1] Nos sistemas distribuídos, o ganho de desempenho pode ser observado através do aumento no *throughput* do sistema. Por exemplo, aumentando-se o número de processadores em servidores de bancos de dados e servidores Web, mais usuários podem ser atendidos simultaneamente. Já nos sistemas paralelos, o ganho de desempenho é mais difícil de ser obtido e está condicionado a questões como: organização dos processadores, linguagem de programação e o grau de paralelismo da aplicação.

[2] *Throughput*: quantidade total de trabalho realizado em um determinado tempo.

[3] Tempo de resposta: também denominado tempo de execução, compreende o intervalo de tempo entre o início e o fim de uma tarefa, ou seja, é o tempo necessário para se completar uma determinada tarefa.

2.1.3 Vantagens e Desvantagens dos Sistemas Paralelos

Os sistemas com múltiplos processadores conservam diversas vantagens, as quais constituem pontos positivos importantes em favor de sua utilização e, conseqüentemente, em detrimento dos sistemas com apenas um processador. As principais vantagens de tais sistemas são [61, 86]:

- **Balanceamento de carga:** a presença deste item nos sistemas distribuídos pode contribuir para a obtenção de melhores resultados, no que concerne o desempenho das aplicações. Isso ocorre porque o balanceamento de carga consiste em distribuir as tarefas a serem executadas entre os processadores da arquitetura, levando-se em consideração a quantidade de trabalho atribuída a cada um e à capacidade de processamento dos mesmos;

- **Escalabilidade**[4] **:** em geral, nos sistemas centralizados, a carga de trabalho de uma máquina cresce até atingir o limite de processamento do hardware, não havendo qualquer maneira de expandi-la. Nesse caso, não há outra solução senão adquirir uma máquina com maior poder de processamento, o que certamente envolverá custos. Nos sistemas distribuídos, o poder de computação pode ser aumentado acrescentando-se, gradualmente, e conforme a necessidade, novos processadores à arquitetura;

- **Melhor aproveitamento de recursos:** fazendo-se uso de arquiteturas distribuídas, na presença de situações onde haja indisponibilidade de recursos locais pode-se, através da rede, utilizar um recurso remoto. Do contrário, seria necessário esperar até que um recurso local estivesse disponível;

- **Melhor aproveitamento de memória:** em computadores convencionais, a quantidade de memória é finita e, muitas vezes, insuficiente, principalmente na resolução de problemas mais complexos. Usando-se a memória de diversos computadores, pode-se superar essa dificuldade com certa facilidade; e

- **Tolerância a falhas e disponibilidade:** se apenas uma máquina está responsável pela operação de uma aplicação e seu processador sofre uma falha, a aplicação inteira é comprometida. Distribuindo-se essa responsabilidade entre diversas máquinas, a falha de uma, embora possa acarretar uma perda

[4]Escalabilidade: é a capacidade de se adicionar novos processadores ao hardware do sistema.

no poder computacional, não afetará o funcionamento geral do sistema. De modo transparente ao usuário, suas tarefas podem ser repassadas a outros processadores, que darão continuidade a sua execução. Sistemas tolerantes a falhas oferecem maior disponibilidade, que é a medida em número de minutos por ano que o sistema permanece em funcionamento de forma ininterrupta, incluindo possíveis falhas de hardware ou software, manutenções preventivas e corretivas. Sistemas de alta disponibilidade são utilizados em aplicações de missão crítica, como sistemas de tráfego aéreo e de comércio eletrônico na Internet [61].

A despeito das inúmeras vantagens dos sistemas distribuídos sobre os sistemas centralizados, existem alguns problemas que acarretam em desvantagens e, por isso, devem ser analisados e considerados na abordagem desse tema. Dentre as desvantagens pode-se citar [61, 86]:

- **Organização dos componentes:** processadores, memórias e periféricos devem ser dispostos de tal maneira a se conseguir uma boa relação custo/desempenho. Essa organização é, muitas vezes, uma tarefa árdua e pode não apresentar bons resultados, fazendo com que o projetista tenha que optar entre custos menores ou melhor desempenho;

- **Redes de comunicação:** a interligação dos computadores através de redes de comunicação implica na necessidade de utilização de softwares para tratamento de mensagens, a fim de que informações não sejam perdidas e comprometam o funcionamento do sistema. A saturação da rede é um segundo problema, gerado pela sobrecarga de mensagens que transitam pela mesma. Nesse caso, é necessário aumentá-la ou substituí-la por uma outra rede com capacidade superior, operação que pode envolver custos adicionais;

- **Segurança:** deve-se realizar medidas para que as informações sejam protegidas contra quaisquer acessos indesejados ou indevidos, que podem interferir no comportamento e na confiabilidade do sistema;

- **Sincronização:** este é um dos maiores problemas da computação distribuída, pois deve haver meios de se garantir a consistência dos dados, em todas as máquinas, e o controle de acesso às diversas posições de memória, a fim de

se impedir que dados desatualizados sejam lidos ou que, mais de um usuário escreva em uma mesma posição de memória simultaneamente;

- **Software:** geralmente, o processo de desenvolvimento de aplicações para sistemas distribuídos é mais trabalhoso, requer maiores conhecimentos de programação e domínio do problema; e

- **Tolerância a falhas:** em alguns sistemas, a tolerância à falhas é dependente não apenas do hardware, mas também do sistema operacional, o que dificulta a sua implementação.

Em geral, entretanto, essas desvantagens são superadas pelas vantagens e resultados apresentados pelos sistemas com múltiplos processadores, o que tem feito com que muitos investimentos sejam aplicados nesse campo.

2.2 Uma Taxonomia de Arquiteturas Distribuídas e Paralelas

Em 1966, Flynn propôs um modelo simples para classificar os computadores, com o objetivo de distinguir as diversas arquiteturas existentes, de acordo com a quantidade de instruções e dados que conseguiam manipular simultaneamente. Utilizado até os dias de hoje, esse modelo simples e fácil de entender, ficou conhecido como Taxonomia de Flynn[5] e teve sua relevância por classificar principalmente as arquiteturas paralelas. Assim, a maior parte dos computadores pode ser incluída em uma dessas quatro (4) categorias [52]:

- *Single instruction stream, single data stream* (SISD):
  - Máquinas com um processador (convencionais);
  - Apenas uma instrução é executada de cada vez;
  - Apenas um conjunto de dados é manipulado de cada vez;
  - Execução determinística;
  - Exemplos: alguns PCs, estações de trabalho com uma CPU e mainframes.

[5] A Taxonomia de Flynn deve ser vista tão somente como uma aproximação da realidade, uma vez que existem máquinas com características pertencentes a mais de uma categoria, e que, portanto, não se enquadram perfeitamente em apenas uma delas.

- *Single instruction stream, multiple data stream* (SIMD)
  - Máquinas paralelas
  - Uma mesma instrução é executada pelos processadores ao mesmo tempo;
  - Conjuntos de dados diferentes podem ser utilizados pelas unidades de processamento;
  - Esse tipo de máquina, tipicamente, tem um despachador de instruções, uma rede interna com largura de banda alta e um vetor grande de pequenas unidades de instruções;
  - Melhor aplicado em problemas específicos, caracterizados pela regularidade, como processamento de imagem;
  - Execução síncrona e determinística;
  - Exemplos:
    * *Processor Arrays*: *Connection Machine CM-2*, Maspar MP-1, MP-2; e
    * *Vector Pipelines*: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820.
- *Multiple instruction stream, single data stream* (MISD)
  - Poucos exemplos dessa classe de computadores paralelos existiram;
  - Exemplos:
    * Filtros de freqüência múltipla operando em um único conjunto de sinais; e
    * Múltiplos algoritmos de criptografia tentando descriptografar uma mensagem codificada.
- *Multiple instruction stream, multiple data stream* (MIMD)
  - O tipo mais comum de computadores paralelos;
  - Cada processador pode executar conjuntos de instruções diferentes;
  - Cada processador pode trabalhar com conjuntos de dados distintos;
  - Execução síncrona ou assíncrona, determinística ou não-determinística;

- Exemplos: conjuntos de computadores paralelos conectados em rede e multiprocessadores simétricos (SMP) - incluindo alguns PCs.

Como citado anteriormente, a categoria MIMD abrange os sistemas com múltiplos processadores e, portanto, é estudada com mais detalhes a seguir.

2.2.1 Classificação de Sistemas MIMD

Por se tratar de um conceito bastante abrangente, as arquiteturas MIMD podem ser classificadas de acordo com o seu grau de acoplamento. Este inclui fatores como número de processadores envolvidos e a distância existente entre os mesmos, organização da memória principal (compartilhamento e tempo de acesso) e estratégia de interconexão (mecanismos de comunicação, sincronização e velocidade de interconexão dos processadores). Assim, os sistemas MIMD podem ser qualificados como sistemas fortemente acoplados ou sistemas fracamente acoplados.

Sistemas Fortemente Acoplados

São sistemas que consistem de um conjunto de processadores que compartilham uma mesma memória principal, e estão sob o controle de um único sistema operacional [82]. Nos sistemas fortemente acoplados, geralmente, todos os componentes estão localizados bem próximos uns aos outros e, freqüentemente, interagem através de redes de comunicação de alta velocidade. Um mesmo espaço de endereçamento é compartilhado, onde a comunicação é feita através de operações de leitura e escrita em variáveis na memória principal [61, 85].

Exemplos: Sistemas com Multiprocessadores Simétricos (SMP) e Sistemas NUMA.

Sistemas com Multiprocessadores Simétricos

Os sistemas com multiprocessadores simétricos (*Symmetric Multiprocessors* - SMP) possuem as características dos sistemas fortemente acoplados. Assim, constituem-se de dois ou mais processadores interligados que compartilham uma mesma memória principal (um único espaço de endereçamento), e são gerenciados por apenas um sistema operacional. Essa arquitetura é referenciada muitas vezes por UMA (*Uniform Memory Access*), pois seus acessos à memória principal são em tempo uniforme, independente da localização física de seus processadores.

Sistemas NUMA

Os sistemas NUMA (*Non-Uniform Memory Access*) também são considerados sistemas fortemente acoplados e, portanto, possuem suas características. São formados pela interligação de dois ou mais conjuntos de componentes através de uma rede de interconexão, onde cada conjunto possui seus processadores, memória e, alternativamente, dispositivos de E/S. A memória de um sistema NUMA é fisicamente distribuída entre os diversos conjuntos de componentes, mas logicamente compartilhada, fornecendo um único espaço de endereçamento. Esse modelo de memória é denominado memória-compartilhada distribuída (ou *Distributed Shared Memory* - DSM) e é abordado neste capítulo. É importante ressaltar que, pelo fato da memória estar fisicamente distribuída, seus tempos de acesso podem variar dependendo da localização do processador. Por isso, tais sistemas são denominados sistemas de acesso não-uniforme à memória (*Non-Uniform Memory Access* - NUMA). Dentro de um mesmo conjunto, os tempos de acesso à memória são menores do que os efetuados a memórias pertencentes a outros conjuntos do sistema.

Sistemas Fracamente Acoplados

Os sistemas fracamente acoplados constituem-se de um conjunto de sistemas computacionais autônomos, conectados por uma rede de comunicação de baixa velocidade. Cada sistema possui seus próprios recursos, como: processadores, memória principal, dispositivos de E/S e sistema operacional. Cada sistema possui seu próprio espaço de endereçamento, onde a comunicação é realizada através de mecanismos de troca de mensagens [61, 82].

Exemplos: *Clusters*, Sistemas Operacionais de Rede (SOR) e Sistemas Distribuídos (SD).

Clusters

São sistemas fracamente acoplados compostos de um conjunto de computadores (também denominados nós), normalmente conectados através de LANs, que se comportam como um grande sistema com múltiplos processadores. Cada nó de computação é um sistema que possui um ou vários processadores (PCs, estações de trabalho, NUMA ou SMPs), memória, dispositivos de E/S e sistema operacional.

Os *clusters* diferenciam-se dos sistemas NUMA e SMP pelo fato de terem um sistema operacional executando em cada um de seus nós, onde a memória é fisicamente distribuída entre os mesmos. Como a classe dos sistemas fracamente acoplados à que pertencem, os nós de computação possuem seu próprio espaço de endereçamento e, geralmente, trocam informações através de mecanismos de passagem de mensagens.

Os *clusters* podem ser utilizados na execução de aplicações seqüenciais e paralelas. Nestas últimas, as tarefas são divididas entre os diversos nós da arquitetura para agilizar o processamento da aplicação e diminuir o seu tempo de execução. Esse processo exige maior envolvimento e conhecimento da arquitetura por parte do programador. Para facilitar a programação de aplicações paralelas em *clusters*, softwares têm sido projetados e desenvolvidos, como o PVM (*Parallel Virtual Machine*) [35], MPI (*Message Passing Interface*) [63] e TreadMarks [4, 50, 68]. Este último é um dos objetos de estudo desse trabalho.

Sistemas Operacionais de Rede

Sistemas Operacionais de Rede (SOR) são sistemas fracamente acoplados, que conectam, administram e mantêm todos os recursos da rede. Constituem-se de um conjunto de sistemas computacionais autônomos (ou nós de computação), conectados por uma rede de comunicação, que pode ser uma rede local (*Ethernet* ou *Token Ring*) ou uma rede distribuída (Internet).

Os nós são sistemas independentes, com espaços de endereçamento individuais e sistemas operacionais heterogêneos, onde a comunicação é feita através de uma interface de rede, utilizando um mesmo protocolo.

Nos SOR não existe a transparência no compartilhamento de recursos, ou seja, não existe o conceito de imagem única do sistema. Os nós conhecem uns aos outros na rede e sabem quais recursos cada um compartilha com os demais. Assim, quando um nó necessita de um componente localizado em outro computador deve acessá-lo para encontrar o recurso compartilhado.

Sistemas Distribuídos

Um sistema distribuído caracteriza-se pela união de duas ou mais estações de trabalho, fisicamente independentes, por meio de uma rede de interconexão, que cooperam no processamento de tarefas, através da execução das pequenas tarefas

distintas que a compõem. Na comunicação é utilizado um mecanismo de troca de mensagens, transparente ao usuário, que contribui para que o sistema comporte-se como um único, conceito este denominado de imagem única do sistema.

A Figura 2.1 apresenta a arquitetura MIMD e sua subdivisão em sistemas fortemente acoplados e sistemas fracamente acoplados, de acordo com a organização lógica de memória, e seus respectivos sistemas representantes.

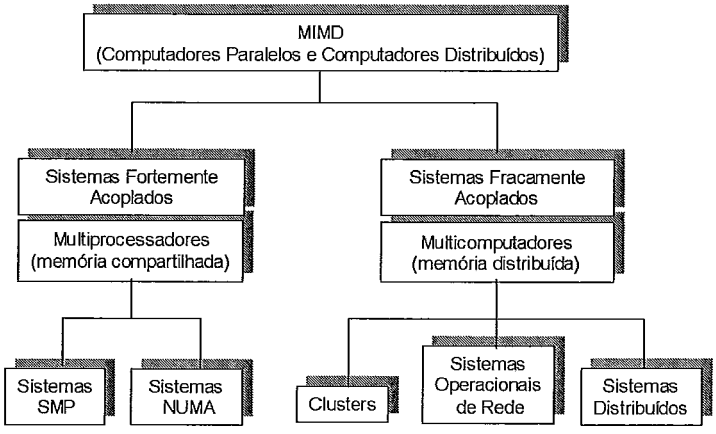


Figura 2.1: Classificação dos Sistemas MIMD

2.3 Organização de Memória

Observando a figura apresentada, podemos dizer que, dependendo do número de processadores utilizados em uma arquitetura, as máquinas MIMD podem ser classificadas e referenciadas logicamente como arquiteturas de memória-compartilhada ou arquiteturas de memória-distribuída.

Na abordagem de organização de memória e de comunicação de dados para essas arquiteturas, é importante ressaltar a diferença entre os conceitos de memória-compartilhada e memória-distribuída quando analisados pelo nível físico e o nível lógico de sua implementação. A organização de memória apresentada neste texto, define a sua disposição física. Já o modelo de comunicação de dados apresenta a disposição lógica, ou seja, o modo como a memória é vista pelos usuários e como os seus dados podem ser comunicados entre os processadores do sistema. A seguir, a abordagem física.

2.3.1 Arquitetura de Memória Centralizada

Em sistemas multiprocessadores (sistemas fortemente acoplados) que possuem um número relativamente pequeno de processadores, é possível implementar uma arquitetura de memória centralizada, onde uma única memória é compartilhada pelos diversos processadores existentes e, conectada a eles, por meio de um barramento. Utilizando-se quantidades grandes de *caches*, a memória e o barramento conseguem atender à demanda de memória dos processadores [43].

2.3.2 Arquitetura de Memória Distribuída

Já em sistemas que devem suportar um número grande de processadores e sua conseqüente demanda por largura de banda de memória, deve-se implementar uma arquitetura de memória distribuída, onde a memória é fisicamente distribuída entre os nós de computação. Cada um destes nós é composto de um ou mais processadores, memória, dispositivos de I/O e uma interface para uma rede de interconexão que conecta todos os nós. Essa arquitetura possibilita um aumento na largura de banda da memória, se a maioria dos acessos é feita à memória local, e diminui a latência nos acessos à mesma. Sua desvantagem está na latência existente no processo de comunicação entre os diversos nós, uma vez que os processadores não mais compartilham uma única memória centralizada. Os sistemas multicomputadores (sistemas fracamente acoplados) implementam a arquitetura de memória distribuída apresentada [43].

2.4 Modelos para Comunicação de Dados e Arquitetura de Memória

Como apresentado anteriormente, sistemas com um número grande de processadores devem implementar uma arquitetura de memória fisicamente distribuída, enquanto sistemas com número relativamente pequeno de processadores podem implementar uma arquitetura de memória fisicamente compartilhada.

Na abordagem de comunicação de dados, que trata da disposição lógica da memória e do processo de comunicação de dados entre os processadores, a memória pode ser classificada como: memória-compartilhada, memória-distribuída ou memória-compartilhada distribuída [43]:

2.4.1 Modelo de Memória-compartilhada

O termo memória-compartilhada se refere ao fato do espaço de endereçamento ser compartilhado, ou seja, o mesmo endereço físico em dois processadores refere-se a uma mesma posição de memória. É importante observar que, memória-compartilhada não implica na existência de apenas uma memória centralizada (também conhecida como UMA), apenas um único espaço de endereçamento compartilhado.

A comunicação de dados em tais máquinas é realizada de modo implícito através de operações de *load/store* sobre o espaço de endereçamento compartilhado.

2.4.2 Modelo de Memória-distribuída

Nesse modelo, o espaço de endereçamento é fisicamente distribuído e privativo para cada processador, ou seja, um mesmo endereço físico em processadores diferentes refere-se a locais distintos em memórias também distintas. Em geral, cada memória/processador pertence a máquinas independentes, chamadas de multicomputadores, onde seus dados não podem ser acessados diretamente por qualquer outra.

A comunicação de dados em máquinas com espaços de endereçamento múltiplos é feita por intermédio de mecanismos explícitos de troca de mensagens entre os processadores. Por isso, essas máquinas são diversas vezes denominadas de máquinas de troca de mensagens.

2.4.3 Modelo de Memória-compartilhada Distribuída

Esse modelo diferencia-se do modelo de memória-compartilhada pelo fato de possuir memórias fisicamente distribuídas. Mas, de maneira similar, podem ser endereçadas virtualmente como um único espaço de endereçamento compartilhado. Desse modo, os processadores compartilham as informações de suas memórias, podendo efetuar uma referência a qualquer um de seus endereços, desde que tenham direito de acesso. Essas máquinas são denominadas de arquiteturas de memória-compartilhada distribuída (*Distributed Shared-Memory* ou DSM) ou arquiteturas escaláveis de memória-compartilhada (*scalable shared-memory architectures*). Uma vez que, o tempo de acesso a um dado depende de sua localização na memória, esses sistemas são também denominados de NUMA.

O modelo de comunicação de dados de arquiteturas DSM é similar ao modelo de memória-distribuída no nível físico, ou seja, através de mecanismos de troca de mensagens e, logicamente, pelo compartilhamento da memória, como ocorre nas arquiteturas de memória centralizada. O modelo DSM é estudado a seguir.

2.5 Conceitos de Memória-compartilhada Distribuída (DSM)

A arquitetura de memória centralizada oferece um modelo de programação simples, onde o compartilhamento de dados é feito por meio de operações de leitura e escrita em áreas da memória-compartilhada, comum a todos os processadores. Todavia, apesar dessa facilidade encontrada na programação, o mesmo não acontece quando o assunto é escalabilidade. Um aumento no número de processadores de um sistema pode acarretar em índices de contenção e latência no acesso à memória, que degradem, de maneira significativa, o desempenho do sistema.

Por outro lado, os sistemas de memória-distribuída são inerentemente escaláveis, ou seja, admitem um número crescente de processadores e, conseqüentemente, são capazes de suportar sistemas que exijam alto poder de computação. Entretanto, seu modelo de comunicação de dados envolve um mecanismo de troca de mensagens que requer o uso explícito de primitivas de *send/receive*. Isso faz com que o programador tenha que gerenciar a comunicação e a distribuição dos dados. Embora esta não seja uma tarefa difícil, apenas mais complexa se comparada ao modelo de memória-compartilhada, requer um maior envolvimento do programador. Além disso, como os nós de computação possuem espaços de endereçamento diferentes, a tarefa de migração de processos acaba sendo também dificultada.

A arquitetura de memória-compartilhada distribuída surgiu, em meados dos anos 80, como proposta para se obter o melhor das arquiteturas de memória-compartilhada e memória-distribuída. Como visto anteriormente, seu modelo de comunicação implementa em nível lógico a abstração do modelo de memória-compartilhada, preservando a facilidade de programação e portabilidade existente nesse modelo. Desse modo, embora fisicamente possua memória-distribuída, essa abstração torna transparente ao programador o mecanismo de troca de mensagens existente nas arquiteturas de memória-distribuída, mantendo a escalabilidade e poder de computação

característico de tais sistemas.

2.5.1 Classificações dos Sistemas de Memória-compartilhada Distribuída

Existem três questões importantes a serem consideradas no projeto e implementação de um sistema de memória-compartilhada distribuída no que diz respeito ao acesso e consistência dos dados:

- **Forma de acesso e sincronização das variáveis (algoritmo DSM);**
- **Nível de implementação do acesso aos dados (nível de implementação do mecanismo da DSM); e**
- **Modelo de consistência de memória (garante a coerência no acesso aos dados)**

Algoritmos de Memória-compartilhada Distribuída (DSM)

Os algoritmos para implementação de DSM lidam com dois problemas básicos [72]:

- Distribuição dinâmica e estática de dados compartilhados pelo sistema, para minimizar a latência de acesso; e
- Preservação de uma visão coerente dos dados compartilhados, minimizando *overheads* no gerenciamento da coerência.

Os algoritmos DSM podem empregar, principalmente, duas estratégias para a distribuição de dados: migração e replicação. A **migração** consiste em permitir que, num determinado momento, apenas um nó possua qualquer dado compartilhado e, portanto, acesso de leitura e/ou escrita sobre o mesmo. Outro nó, que deseje acessá-lo, deverá fazer uma solicitação e esperar até que o dado seja migrado e recebido, passando a ter, sobre o mesmo, acesso exclusivo. A **replicação** consiste em permitir que diversos nós possam ter uma cópia de um dado compartilhado, simultaneamente. Em geral, essa estratégia é utilizada para permitir que vários nós possam ter acesso de leitura a um dado, de forma concomitante. Assim, dependendo dos tipos de acesso à memória que devem ser permitidos e das configurações que o sistema deve possuir, uma estratégia prefere a outra.

Algoritmos de um leitor/um escritor (*single reader/single writer* - SRSW)

Nesse algoritmo, não se pode usar a estratégia de replicação, uma vez que só deve haver uma cópia do item de dado em qualquer instante, seja para leitura ou para escrita. O algoritmo de gerenciamento de DSM mais simples desse tipo denomina-se algoritmo com servidor centralizado (*central server algorithm*). Nele, um nó assume o papel de servidor, concentrando em si toda a memória-compartilhada do sistema, e recebendo/atendendo a todas as solicitações. O problema desse algoritmo está no gargalo gerado no servidor e na conseqüente queda no desempenho, pois todas as informações compartilhadas são mantidas e acessadas em um único nó. Uma solução para esse problema é distribuir, estaticamente, a memória física entre diversos servidores, de forma que cada um fique responsável pelo gerenciamento de acesso em sua parcela de memória-compartilhada.

Alguns algoritmos SRSW mais sofisticados permitem a migração dos dados e são denominados de *hot potato* (batata-quente). Uma vez que, os dados são movidos em unidades de tamanho fixo (blocos) e não individualmente, se o princípio da localidade se aplicar em sistemas com tais algoritmos, o custo da migração de dados pode ser amortizado. O desempenho pode melhorar, também, se: inúmeros acessos seguidos forem efetuados por um único nó sem a interrupção de outros e, se houver escrita após leitura a um mesmo dado, freqüentemente. De qualquer forma, em geral, o desempenho desse algoritmo é baixo, porque não consegue explorar o potencial de aplicações paralelas onde há predominância de múltiplos acessos de leitura. Para tais aplicações foram desenvolvidos os algoritmos a seguir.

Algoritmos de múltiplos leitores/um escritor (*multiple reader/single writer* - MRSW)

O objetivo principal desses algoritmos é reduzir o custo médio de operações de leitura, predominantes em aplicações paralelas. Assim, permitem que diversos nós possam compartilhar, simultaneamente, um determinado dado para leitura, através da estratégia de replicação. Entretanto, na operação de escrita o nó deve ter acesso exclusivo ao dado, o que normalmente é feito utilizando-se um protocolo de invalidação, onde as cópias residentes em outros nós são invalidadas. Esse procedimento acarreta em custo adicional para a operação de escrita.

Os algoritmos dessa classe diferenciam-se pela maneira como é distribuída a responsabilidade pelo gerenciamento no acesso à DSM. Diversos algoritmos foram

propostos por LI e HUDAK [55], incluindo-se: *Centralized manager algorithm*, *Improved centralized manager algorithm*, *Fixed distributed manager algorithm*, *Broadcast distributed manager algorithm* e *Dynamic distributed manager algorithm*. Um resumo do funcionamento de cada um deles pode ser encontrado em [72].

Algoritmos de múltiplos leitores/múltiplos escritores *(multiple reader/mulitple writer* - MRMW)

Os algoritmos de múltiplos leitores/múltiplos escritores (MRMW) permitem a replicação dos blocos de dados com ambas as permissões: de leitura e de escrita. Quando um nó efetua uma operação de escrita, deve enviar uma atualização da cópia para todos os outros nós, através de mensagens de *broadcast* ou *multicast*. Normalmente, esse procedimento é feito empregando-se um protocolo de atualização (*write-update protocol*), por meio do qual os algoritmos MRMW tentam minimizar o custo dos acessos de escrita e permitir que haja múltiplos escritores para as áreas de dados compartilhadas de um sistema. Como desvantagem, algoritmos MRMW podem produzir altos índices de tráfego de coerência se a freqüência de atualizações e o número de cópias replicadas forem expressivos.

A implementação de *multicast* de forma confiável é uma técnica alternativa, aplicada para manter os dados do sistema consistentes, através da seriação de todas as suas operações de escrita. Utiliza-se um mecanismo denominado seqüenciador, cuja função é: receber as solicitações de atualização em posições de memória-compartilhada e, enviar uma mensagem de *multicast*, com tal informação, para todos os outros nós do sistema que possuam uma cópia do dado. Um número seqüencial é designado, a cada uma dessas mensagens, e verificado, quando de sua recepção pelos nós, que solicitam a retransmissão dos dados caso seu valor esteja incorreto.

Uma modificação deste algoritmo distribui a tarefa realizada pelo seqüenciador. Nesta solução, a responsabilidade pela consistência das estruturas de dados é distribuída entre os nós, que devem aplicar as funções do seqüenciador às estruturas sob seu encargo. Nesse esquema, muito embora o sistema não esteja coerente seqüencialmente, cada estrutura de dados é mantida coerente.

Nível de Implementação do Mecanismo de DSM

Uma das decisões mais importantes, no projeto e implementação de sistemas DSM, é determinar o nível em que o mecanismo de DSM será implementado, o que pode afetar o desempenho, a programação e aumentar os custos totais do sistema.

O modelo de DSM foi apresentado, anteriormente, como tendo surgido para extrair o que há de vantajoso nas arquiteturas de memória-compartilhada e memória-distribuída. Assim, viu-se que o mesmo implementa o modelo de memória-compartilhada em nível lógico, para preservar a facilidade de programação, e o modelo de memória-distribuída em nível físico, para obter escalabilidade e maior poder de computação. Nesse contexto, seu trabalho concentra-se em duas tarefas principais:

1. Supervisionar os acessos efetuados à memória-compartilhada. Nesse momento, sua função é verificar se os endereços desejados encontram-se na memória local. Os que não se encontrarem localmente, devem ser buscados em outros nós e seus dados trazidos para a memória local; e

2. Manter a coerência dos dados compartilhados nos acessos de leitura e escrita, de forma que os diversos nós do sistema tenham acesso aos valores atualizados dos dados.

Essas tarefas podem ser implementadas em nível de software, hardware ou uma combinação de ambos. Geralmente, essa escolha depende de alguns fatores, como a relação custo/desempenho intrínseca de cada implementação. O mecanismo DSM aplicado no nível físico, por exemplo, oferece melhores resultados no desempenho, mas requer a utilização de hardware adicional, muitas vezes suportado apenas por máquinas de grande porte. As demais ficam restritas às implementações em software, e se for possível adicionar um hardware de baixo custo, a uma solução híbrida, como é visto a seguir.

Implementações do Mecanismo de DSM em Nível de Software

Até a década de 1980, os sistemas de memória-distribuída utilizavam apenas um modelo de comunicação baseado no mecanismo de troca de mensagens. Nele, os programadores eram obrigados a se preocupar com a distribuição dos dados e com o seu processo de envio para os outros nós, ou seja, em requisitar que o sistema fizesse

a sua transmissão por meio de mensagens na rede. Além disso, nesses sistemas, a transmissão de estruturas de dados complexas e a migração de processos introduziam problemas adicionais. Foi nesse contexto, que começou a se disseminar a idéia de construir uma camada de software, que fornecesse o paradigma do modelo de memória-compartilhada, sobre a camada do mecanismo de troca de mensagens. Assim, a programação seria facilitada e se manteria a escalabilidade e poder de computação dos sistemas distribuídos.

Para realizar a implementação, essa camada pode ser adicionada em nível de usuário, em rotinas de bibliotecas de *run-time*, no sistema operacional ou numa linguagem de programação. O suporte em software para DSM é geralmente mais flexível do que o feito em hardware, e os mecanismos de consistência são mais facilmente adaptáveis ao comportamento da aplicação. Entretanto, softwares DSM perdem em desempenho para implementações em hardware.

Alguns exemplos de implementações de software DSM: IVY [54], Mermaid [91], Munin [19], TreadMarks [4, 50, 68], Midway [11], Blizzard [77], Mirage [33], Clouds [73], Orca [8], Linda [1] e HLRC [48, 75, 76].

Implementações do Mecanismo de DSM em Nível de Hardware

Mecanismos DSM implementados em hardware garantem a replicação automática dos dados compartilhados nas memórias locais, assim como nas *caches* dos processadores, de forma transparente às camadas de software. Esta abordagem suporta de forma eficiente o compartilhamento de granulosidade fina. A unidade física e não-estruturada de replicação e de coerência é pequena, tipicamente, uma linha de *cache*. Conseqüentemente, os mecanismos de hardware DSM, usualmente, representam uma extensão dos princípios encontrados em esquemas de coerência de *cache*, presentes em arquiteturas escaláveis de memória-compartilhada. Esta abordagem reduz, de forma significativa, os requisitos de comunicação, porque granulosidades mais finas minimizam os efeitos adversos do falso compartilhamento e *thrashing*. As funções de diretório e de busca implementadas em hardware são mais rápidas do que as implementações em nível de software, diminuindo as latências de acesso à memória. Todavia, técnicas avançadas de manutenção de coerência e de redução de latência, normalmente, complicam o projeto e a verificação. Por isso, um hardware DSM é freqüentemente utilizado em máquinas onde o desempenho é mais importante do que o custo.

Dependendo da arquitetura do sistema de memória, três grupos de sistemas de hardware DSM são especialmente interessantes:

- *cache coherent non-uniform memory architectures* (CC-NUMA): Memnet [28], Dash [53] e SCI [49];
- *cache-only memory architectures* (COMA): KSR1 [34] e DDM [42];
- *reflective memory system* (RMS): RMS [60] e Merlin [62].

Implementações do Mecanismo de DSM em Nível Híbrido

Os estudos realizados em implementações do mecanismo de DSM, em nível de software e hardware, mostraram que estas abordagens não ficavam totalmente restritas ao nível a que se propunham atender. Ademais, nenhuma das duas concentrava todas as vantagens em si. Portanto, começaram a surgir modelos híbridos, com elementos de software e de hardware combinados de forma parcial ou predominante, para balancear o custo de complexidade.

Alguns exemplos de implementações híbridas de DSM: Plus [15], Galactica Net [90], MIT Alewife [20], Flash [51] e NCP2 [12].

Modelos de Consistência de Memória

Em sistemas DSM, os nós se comunicam por meio de variáveis compartilhadas, onde os acessos de leitura e escrita devem ser ordenados, de tal forma que, a aplicação apresente um comportamento determinado. Assim, surgiram os modelos de consistência de memória, com a proposta de manter a memória global do sistema coerente. A **consistência** define a ordem das referências efetuadas pelos processadores à memória-compartilhada, e se diferencia da **coerência**, pelo fato desta última definir a ordem dos acessos realizados à memória local [43]. Aplicações de tipos diferentes podem se adequar melhor a modelos de consistência distintos, os quais interferem no comportamento do sistema, alterando fatores como o desempenho, os tempos de acesso e requisitos de largura de banda do sistema [72].

Exemplos de modelos de consistência de memória:

- *Sequential consistency*: nesse modelo, todos as operações de leitura e escrita devem ser ordenadas como quando executadas em apenas um processador. Exemplos de sistemas DSM: IVY [54] e Mirage [33];

- *Processor consistency*: nesse modelo, a ordem na qual diferentes processadores podem ver as operações de memória não precisa ser idêntica, mas todos os processadores devem observar a seqüência de escritas de cada processador na mesma ordem. Exemplos de sistemas DSM: Plus [15], Merlin [62] e RMS [60];

- *Weak consistency*: a consistência da memória é realizada apenas nas operações de sincronização, estas baseadas no modelo seqüencial e distintas, agora, dos outros acessos comuns de leitura e de escrita. Uma operação de sincronização está condicionada ao término de todas as outras operações anteriores, do mesmo modo que leituras e escritas devem esperar pelo término dos acessos anteriores de sincronização;

- *Release consistency*: os acessos de sincronização são divididos em operações de *acquire* e *release*, que devem proteger os acessos a áreas de memória-compartilhada. Assim, um processo faz um *acquire* e tem sua memória coerente para os dados no escopo relativo àquela operação. De forma similar, ao final das operações de leitura e escritas desejadas nesses dados, o processo deve liberá-los, realizando uma operação de *release*, quando os novos valores serão atualizados nos outros nós. Exemplos de sistemas DSM: Dash [53] e Munin [19]; e

- *Lazy release consistency*: similar ao modelo *Release Consistency*, ou seja, com duas operações de sincronização (*acquire* e *release*) que mantêm a memória consistente. Entretanto, esta sincronização não mais é feita no momento do *release* e sim no *acquire* seguinte, quando já se conhece o nó que está solicitando as informações atualizadas. Com isso, diminui-se a quantidade de transferência de dados, que são enviados a um único nó e apenas aqueles relativos ao *lock* específico. Exemplo de sistema: TreadMarks [4, 50, 68] e HLRC [48, 75, 76].

2.6 TreadMarks: Um Exemplo de Software DSM

TreadMarks é um sistema DSM cuja proposta é permitir que, aplicações paralelas baseadas no modelo de memória-compartilhada, sejam executadas em uma rede de estações de trabalho, ou seja, em hardware que suporta o mecanismo de troca de mensagens [68]. Foi totalmente implementado como uma biblioteca no nível de usuário de sistemas Unix, sem requisitar modificações no *kernel*, uma vez que

tais sistemas provêem todas as funções necessárias de comunicação e gerenciamento de memória. Não requer privilégios especiais de acesso e combina-se bem com compiladores, *link*-editores e interfaces padrão do Unix. Assim, programas escritos em C, C++ ou Fortran podem ser compilados e *link*-editados com a biblioteca do TreadMarks por meio de qualquer compilador padrão para a linguagem. A combinação desses fatores resultou na portabilidade do TreadMarks para diversas outras plataformas, como IBM RS-6000, SP-1 e SP-2; DEC Alpha e DEC-Station, assim como sistemas Sun, Hewlett-Packard e Silicon Graphics [4].

2.6.1 Projeto

Dois aspectos importantes no projeto de um sistema DSM estão na escolha do modelo de consistência e estrutura de memória utilizados. O TreadMarks provê a memória-compartilhada como um vetor linear de bytes e utiliza uma variação do modelo de consistência *Release Consistency*.

O projeto do TreadMarks se concentra em reduzir a quantidade de comunicação necessária para manter a coerência da memória. Para tanto, utiliza o modelo de consistência *Lazy Release Consistency*, que lida com a sincronização e tenta diminuir o número de mensagens e quantidade de dados na rede, e um protocolo de múltiplos escritores para tratar o problema do **falso compartilhamento**. Este ocorre quando dois ou mais processos tentam acessar variáveis diferentes dentro de uma mesma página, com pelo menos um dos acessos sendo de escrita. Protocolos de múltiplos escritores requerem a criação de *diffs*, estruturas de dados que registram as atualizações a partes de uma página. Com o modelo *Lazy Release Consistency*, a criação de *diffs* pode ser adiada ou evitada, uma técnica conhecida como *lazy diff creation*.

2.6.2 Sincronização

Em sistemas DSM, as variáveis contidas na área de memória-compartilhada são vistas e acessadas por todos os processos pertencentes a uma mesma aplicação. Nesse contexto, pode acontecer de dois ou mais processos tentarem acessar uma mesma variável simultaneamente. Se pelo menos um desses acessos for uma operação de escrita, é gerada uma condição de corrida (*data race*), e o sistema pode não ter o comportamento e resultados esperados. Por exemplo, se um processo estiver atualizando um registro e um outro tentar ler algum de seus campos, pode acabar

recebendo uma parte dos dados atualizada e outra ainda desatualizada. Para evitar esse tipo de problema, é preciso que haja algum mecanismo de sincronização que controle os acessos de leitura e escrita das variáveis compartilhadas. O TreadMarks provê duas primitivas de sincronização:

- *Locks*: suportam duas operações: *acquire* e *release*. A operação de *acquire* obtém controle de um *lock* para um único processo. A operação de *release* libera o controle de um *lock* [68].
- Barreiras: impedem o progresso de um processo até que todos os outros alcancem a barreira designada [68].

Estas primitivas são estudadas em maiores detalhes no Capítulo 4.

2.6.3 Modelo de Consistência de Memória Lazy Release Consistency (LRC)

A sincronização é importante para impedir que um processo acesse uma área de memória que esteja sendo atualizada por um outro nó do sistema. Entretanto, existe um ponto a ser considerado: o momento em que essa atualização deve ser informada a outro processo. O TreadMarks utiliza o modelo *Lazy Release Consistency* (LRC), um modelo de consistência relaxado de memória, que permite a um processo postergar a propagação de suas atualizações em dados compartilhados até que uma determinada operação de sincronização ocorra. Os acessos à memória-compartilhada podem ser de sincronização ou considerados normais. Os acessos de sincronização dividem-se em: *acquire* e *release*. O primeiro corresponde a uma operação de *lock*, enquanto o segundo corresponde a uma operação de *unlock*. Assim, as alterações de um processo p na memória-compartilhada (que deve ser liberada por p através de uma operação de *release*), são visíveis a um processo q apenas quando este efetuar o seu próximo acesso de sincronização, ou seja, o *acquire* correspondente ao *release* de p.

Um aspecto importante a ser ressaltado é a distinção existente entre uma operação que apenas comunica que uma determinada página foi atualizada e uma outra que realmente propaga o seu conteúdo, ou seja, os valores alterados. O momento em que estas operações são realizadas depende do protocolo utilizado pelo sistema. O TreadMarks utiliza juntamente com o LRC, um protocolo de invalidação.

Por ele, no momento da sincronização, as páginas alteradas são apenas invalidadas. Um acesso posterior causará um *access miss*, que fará com que um *diff* da página, invalidada anteriormente, seja trazido para o nó e aplicado à página para que esta se torne atualizada. Resumidamente, o LRC determina que a sincronização deve ser postergada para o momento de um *acquire* e não no momento em que a modificação ocorre (*release* anterior). E o protocolo de invalidação determina que as páginas atualizadas devem ser invalidadas e seu conteúdo atualizado aplicando-se os *diffs* recebido posteriormente quando forem acessadas e causarem um *access miss* no sistema.

2.6.4 Protocolos de múltiplos escritores

Os sistemas que implementam memória-compartilhada devem, em seus projetos, se preocupar com gerenciamento da memória no sentido de tratar o problema de falso compartilhamento. Em protocolos que permitem apenas um escritor (*single writer*), antes que uma página possa ser escrita, todas as suas cópias em outros processos devem ser invalidadas. Como consequência, ocorrerá um *access miss* em todos os processos que tentarem acessar essa página, forçando-os a buscar na rede uma nova cópia atualizada. Acontece, porém, que esse procedimento é desnecessário, uma vez que a variável escrita não está sendo acessada pelos demais processos. Portanto, os mesmos não precisam da informação de sua atualização. Conclui-se, desse modo, que este protocolo (*single writer*) incorre em excesso de comunicação, pois dois processos podem estar acessando variáveis distintas e ainda assim são obrigados a sincronizar suas páginas.

Para resolver esse problema, foi criado o protocolo de múltiplos escritores, onde dois ou mais processos podem ter, ao mesmo tempo, uma cópia de escrita de uma página, desde que em posições distintas. Nesse caso, como as modificações de uma página são capturadas?

2.6.5 Comunicação entre Processos e Criação de Lazy Diffs

TreadMarks implementa a comunicação entre processos usando *sockets* UDP/IP (*User Datagram Protocol/Internet Protocol*) de Berkeley, protocolos no nível de usuário para garantir a entrega das mensagens e primitivas bloqueantes para a sincronização. Toda mensagem enviada pelo TreadMarks, é uma requisição ou uma resposta. As mensagens, contendo requisições, são enviadas ou como resultado

de uma chamada explícita a uma rotina de sua biblioteca, ou devido a um *page fault*. Em seguida, o processo deve permanecer bloqueado até que chegue uma outra mensagem de requisição ou a resposta desejada. Se, após um determinado tempo, esta última não for recebida, a requisição original deve ser retransmitida. Com o objetivo de minimizar o atraso gerado na tarefa de manipular as mensagens de requisições, o TreadMarks usa um *handler* do sinal `SIGIO`. Assim, a chegada de uma mensagem em um *socket*, usado para receber requisições, gera um sinal `SIGIO`. O *handler*, então, realiza a operação especificada, envia uma mensagem de resposta, se houver necessidade, e retorna ao processo que foi interrompido [4].

Para implementar o protocolo de consistência, o TreadMarks usa a chamada de sistema `mprotect`, que controla os acessos a páginas compartilhadas, gerando um sinal `SIGSEGV` na detecção de qualquer tentativa de acessos restritos. Nesse caso, o *handler* correspondente examina as estruturas de dados locais, para determinar o estado da página. Se a mesma é inválida, obtém os *diffs* necessários do conjunto mínimo de máquinas remotas. Em seguida, verifica a pilha de exceção para determinar se a referência é uma leitura ou uma escrita. Na primeira, a proteção da página é configurada para somente-leitura. Na segunda, cria um *twin* a partir do *pool* de páginas livres. A mesma ação é tomada em resposta a uma falha causada por uma escrita numa página no modo somente-leitura. Finalmente, o *handler* altera os direitos de acesso para a página original e devolve o controle da execução para a aplicação [4].

Entendido o mecanismo de comunicação entre os processos e a implementação do protocolo de consistência, resta verificar como ocorre o processo de criação de *diffs*. Veja um exemplo de como todo esse processo é feito [4]: Considere dois processos P1 e P2 que escrevem de forma concorrente em diferentes posições dentro de uma mesma página. Ambos inicialmente possuem uma cópia válida, que está protegida contra escrita. Quando P1 tenta escrever nesta página, o hardware de memória virtual detecta uma violação de proteção. O software DSM, então, captura esta violação, cria uma cópia (ou *twin*) da página violada em P1, e remove a proteção de escrita no espaço de endereçamento do usuário, de forma que outras escritas possam ocorrer sem a intervenção do software [4, 50].

O *twin* e a cópia atual podem a partir de então serem comparados para a criação de um *diff*, um *runlength encoded record* das modificações nas páginas. Mas, é importante ressaltar que, no TreadMarks, *diffs* são criados apenas quando um

processo solicita as modificações de uma página ou quando chega uma notificação de escrita de outro processo para a mesma. Neste último caso, é essencial fazer um *diff* para distinguir as modificações locais das realizadas pelos outros processos [4, 50].

No exemplo hipotético, quando P1 chegar em uma barreira, haverá uma cópia modificada e um *twin* intacto. Uma comparação (palavra por palavra) criará um *diff*, ou seja, um *run-length encoding* das modificações na página. Apenas quando o *diff* for criado, o *twin* será descartado. A mesma seqüência de eventos ocorrerá em P2 [4]. Como esses eventos são locais a cada processo, eles não têm as exigências de comunicação de um protocolo *single writer*. Quando P1 e P2 sincronizarem (através de uma barreira, por exemplo), P1 será informado de que P2 modificou a página, e vice-versa, o que fará com que ambos invalidem suas cópias. Quando posteriormente acessarem a página, ocorrerá um *acess fault*. O software de TreadMarks em P1, que terá registrado P2 como modificador da página, enviará uma mensagem a ele solicitando o *diff*, aplicando-o em seguida à página. A mesma seqüência de eventos ocorrerá em P2 [4].

Desse exemplo, pode-se observar que, exceto para os acessos iniciais, as páginas são atualizadas exclusivamente pela aplicação de *diffs*, e cópias completas não são necessárias.

Os protocolos de múltiplos escritores podem ser implementados através dos conceitos de *twins* e *diffs*, reduzindo-se, assim, os efeitos do falso compartilhamento. Além disso, *diffs* reduzem de forma significativa os requisitos gerais de largura de banda, porque podem ser bem menores do que uma página.

2.6.6 Interface de Programação do TreadMarks

A API (*Application Program Interface*) do TreadMarks provê facilidades para a criação e destruição de processos, sincronização, alocação e liberação de memória-compartilhada. No apêndice B, é apresentada a interface fornecida pelo TreadMarks.

2.7 Paralelismo em Programação em Lógica

Para que o processamento de uma aplicação possa ocorrer de forma paralela, a linguagem de programação deve fornecer os meios para se [69]:

1. **Identificar o paralelismo pelo reconhecimento dos componentes da execução do programa que podem ser executados por diferentes processadores;**

2. **Iniciar e terminar a execução paralela;**

3. **Coordenar a execução paralela, especificando e implementando as interações existentes entre componentes concorrentes.**

Esses ambientes de programação podem usar estruturas explícitas que permitem ao programador definir os pontos de paralelismo de um programa, ou explorá-lo automaticamente sem a interferência do programador. Essas duas abordagens são chamadas, respectivamente, de paralelismo explícito e paralelismo implícito.

O **paralelismo explícito** é caracterizado pela presença de construções explícitas na linguagem de programação, que descrevem com alguns detalhes a maneira pela qual a computação paralela ocorrerá. Tem como principal vantagem a flexibilidade na implementação do paralelismo, permitindo ao programador determinar o que será executado em paralelo e o procedimento para que isso ocorra.

O **paralelismo implícito** permite aos programadores escrever suas aplicações sem qualquer preocupação com a questão do paralelismo. Nessa abordagem, o paralelismo é transparente ao programador porque é automaticamente detectado pelo compilador e/ou sistema de execução. Sua principal vantagem do paralelismo implícito está em retirar do programador a responsabilidade de gerenciar o paralelismo e transferi-la para sistemas específicos, destinados a administrar a complexidade de realizar essa tarefa. Sua desvantagem se torna evidente quando se tenta paralelizar aplicações inerentemente seqüenciais ou com paralelismo *fine-grained*, conduzindo geralmente a execuções ineficientes.

O processamento paralelo é explorado, de modo explícito ou implícito, dependendo da abordagem utilizada, que é basicamente composta por três (3) tipos:

- **Linguagens de programação seqüenciais (imperativas) estendidas com construções que permitam a obtenção de paralelismo.** Esse modelo de programação torna mais difícil a tarefa de desenvolver aplicações, uma vez que o programador está envolvido diretamente com o gerenciamento do paralelismo [37];

- **Compiladores que automaticamente paralelizem programas seqüenciais.** Essa tarefa também é bastante difícil e, em geral, não consegue explorar todo o paralelismo disponível no programa [37]; e

- **Linguagens declarativas.** Possuem um modelo de programação considerado de mais alto nível se comparado ao das linguagens imperativas, onde o programador não precisa se preocupar com o gerenciamento do paralelismo. Assim, em linguagens desse tipo, como as linguagens de programação em lógica, o paralelismo pode ser explorado de maneira implícita, tornando os programas mais concisos e reduzindo seus custos [27, 36].

A programação em lógica é um paradigma baseado em um subconjunto da lógica de predicados de primeira ordem, formado por cláusulas de Horn[6] [58]. Surgiu nos anos 70, a partir de estudos realizados sobre o cálculo de predicados na área de lógica formal matemática. É utilizada em aplicações de diversas áreas como Inteligência Artificial, banco de dados, seqüenciamento genético, sistemas especialistas, processamento de linguagens naturais, prova de teoremas, assim como na programação de propósito geral e resolução de problemas [37].

Dentre as diversas linguagens existentes de programação em lógica, a linguagem Prolog é a mais conhecida e difundida na área, sendo desde 1995, definida pelo padrão ISO [29]. Foi criada no início dos anos 70 por Alain Colmerauer, e sua equipe, na Universidade de Marseille. Inicialmente implementada através de interpretadores, com tempos de execução muito altos, posteriormente foi compilada para uma linguagem de baixo nível. Em 1983, os princípios concebidos nessa experiência deram origem a uma máquina virtual denominada *Warren Abstract Machine* [7], ou simplesmente WAM, desenvolvida por David H. D. Warren em Edimburgo. Utilizada nas implementações mais recentes, a máquina abstrata WAM consiste de uma arquitetura de memória e um conjunto de instruções adaptado ao Prolog, sendo suportada por diversos hardwares e constituindo um padrão para o desenvolvimento de compiladores Prolog. Maiores informações sobre a linguagem Prolog podem ser encontradas em [6, 22, 23, 24, 66, 83, 84].

A programação em Prolog consiste de [22]:

- **Declaração de alguns fatos sobre objetos e seus relacionamentos;**

[6]Cláusulas que possuem, no máximo, um literal positivo como conseqüente da implicação.

- **Definição de algumas regras sobre objetos e seus relacionamentos; e**
- **Solicitação de consultas sobre objetos e seus relacionamentos.**

De forma resumida, pode-se dizer, então, que Prolog é uma linguagem de programação usada para resolver problemas que envolvem objetos e seus relacionamentos. É composta por um conjunto de cláusulas de Horn, que expressam condições e conclusões sobre os objetos. Um programa Prolog consiste de dois tipos de cláusulas: regras e fatos.

- **Regras**: são cláusulas que expressam relações condicionais, ou seja, que podem ser verdadeiras ou falsas dependendo de outras relações. Apresentam-se da seguinte forma: p(terms) :- q1(terms),... , qn(terms).

  Onde **p(terms)** é o termo que representa a conclusão (cabeça) da cláusula.

  E **q1(terms), ... , qn(terms)** é um conjunto de termos que representam a condição da cláusula, também denominados de predicados, literais ou objetivos (*goals*). Devem ser provados individualmente como verdadeiros para que a cláusula tenha sucesso e assim se possa concluir p(terms) também como verdadeiro.

  O símbolo **:-** separa a conclusão (cabeça) das condições (corpo) da cláusula.

- **Fatos**: são cláusulas que expressam tautologias, ou seja, relações sempre verdadeiras (axiomas). São representados na forma: p(terms). Pode-se considerar, também, que fatos são cláusulas sem corpo, ou seja, não dependem de nenhuma condição. Desse modo, são incondicionalmente verdadeiros, constituindo, portanto, uma tautologia.

Existem ainda regras especiais denominadas **consultas** (cláusulas sem cabeça), que são sempre executadas e constituem-se de uma conjunção de termos da forma: ?- q1(terms), ... , qm(terms).

O processo de computação consiste em selecionar cada um dos termos (ou *goals*) da consulta e unificar seus argumentos com aqueles da cabeça de alguma das cláusulas do programa. Se a unificação suceder, substitui-se o objetivo selecionado pelos objetivos do corpo da cláusula encontrada, e assim sucessivamente, até que não restem mais objetivos para serem executados (provados).

As Figuras 2.2 e 2.3 apresentam um exemplo simples de um código Prolog composto de fatos, regras e consultas. Nele, são apresentadas as condições para que uma pessoa seja presidenciável no Brasil. De uma forma simplificada, considere que para ser presidenciável uma pessoa precise preencher duas condições: ser brasileira nata e ter pelo menos 35 anos. O código do programa em Prolog define essa condição com a regra presidenciavel/1[7], que possui dois objetivos: brasileiro_nato/1 e idade_35/1. O primeiro é definido por três regras, onde é considerado brasileiro nato aquele que nasceu no Brasil ou que possui pai ou mãe nascidos no Brasil.

```
01.  nasceu(tatiana,brasil).
02.  nasceu(joao,brasil).
03.  nasceu(maria,brasil).
04.
05.  idade(tatiana,26).
06.  idade(joao,36).
07.  idade(david,41).
08.  mae(david,maria).
09.
10.  presidenciavel(Pessoa) :- brasileiro_nato(Pessoa), idade_35(Pessoa).
11.  brasileiro_nato(Pessoa) :- nasceu(Pessoa,brasil).
12.  brasileiro_nato(Pessoa) :- pai(Pessoa,X), nasceu(X,brasil).
13.  brasileiro_nato(Pessoa) :- mae(Pessoa,X), nasceu(X,brasil).
14.  idade_35(Pessoa) :- idade(Pessoa,Y), Y>35.
15.
16.  ?- presidenciavel(tatiana).
17.  ?- presidenciavel(joao).
18.  ?- presidenciavel(david).
19.  ?- mae(david,X).
```

Respostas:
no
yes
yes
yes maria

Figura 2.2: Exemplo de um programa em Prolog

2.7.1 Prolog como Linguagem de Programação em Paralelo

A programação em lógica compreende modelos de execução que exploram o paralelismo de forma explícita e implícita, mas em geral são utilizados os modelos implícitos, devido às facilidades apresentadas anteriormente. Na linguagem Prolog, identifica-se dois tipos principais de paralelismo: OU e E.

O **paralelismo OU** consiste na execução paralela de regras que constituem uma determinada relação. Ocorre quando um objetivo pode unificar com a cabeça de mais de uma cláusula. Nesse caso, cada grupo de objetivos no corpo dessas cláusulas pode ser executado em paralelo com os demais grupos[41]. Examinando o programa da Figura 2.2, o paralelismo OU consiste na exploração em paralelo das

[7]predicado/número é uma notação sintática para representar um predicado e sua aridade, ou seja, seu número de argumentos.

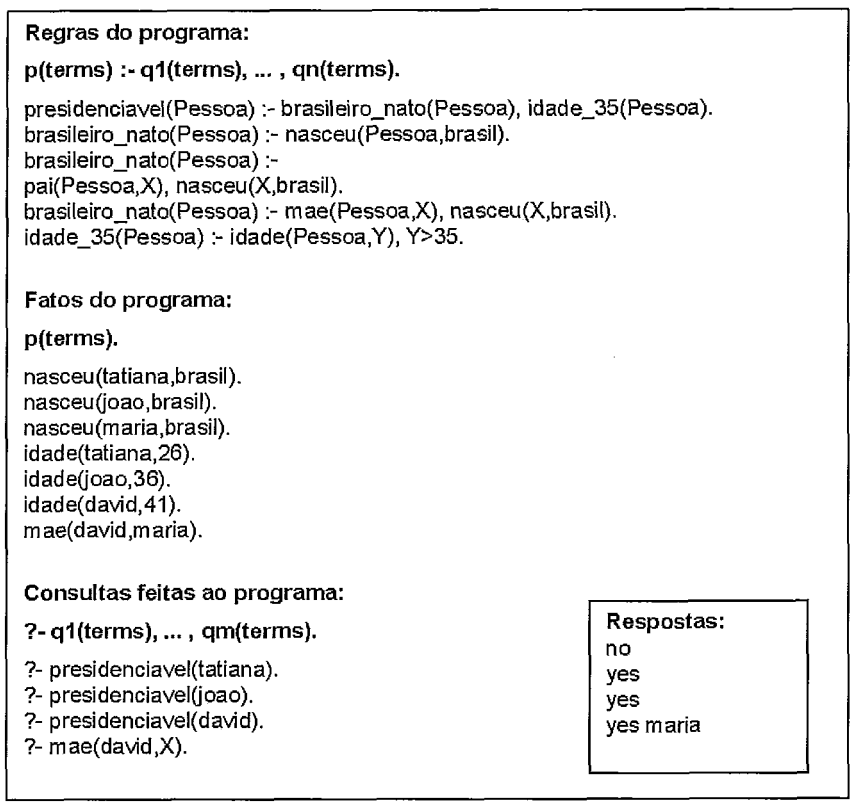


Figura 2.3: Regras, fatos e respostas às consultas feitas ao programa

três regras alternativas correspondentes às linhas 11, 12 e 13, quando estas podem ser disparadas pela execução do objetivo brasileiro_nato/1, na cláusula da linha 10.

No processamento seqüencial, essas regras são executadas uma a uma, e o sucesso (encontro de uma resposta) de uma delas corresponde ao sucesso do objetivo oriundo da consulta, que a elas foi unificado. A expressão em lógica matemática correspondente a essa situação pode ser representada pelas diversas cláusulas (pertencentes a uma mesma relação) conectadas por um operador OU ($\vee$). O paralelismo OU foi assim designado a partir dessa expressão matemática.

Alguns exemplos de sistemas que exploram paralelismo OU: Aurora [32, 31] e MUSE [2, 3].

O **Paralelismo E** consiste na execução paralela de diversos objetivos dentro de uma mesma cláusula. Nesse caso, cada objetivo deve suceder para que a cláusula inteira tenha sucesso, ou seja, que encontre uma resposta para a consulta efetuada. Novamente, a expressão matemática correspondente a essa situação pode ser representada pela conexão desses objetivos através de um operador E ($\wedge$), e por isso, mais uma vez, a origem do nome dado ao paralelismo explorado dessa forma.

O paralelismo E pode ser classificado como independente ou dependente. O

paralelismo E independente é a forma de paralelismo E explorada quando os objetivos pertencentes à mesma cláusula não compartilham variáveis, ou seja, não dependem da execução de qualquer um dos outros para dar continuidade ao seu processamento.

Considere o programa da Figura 2.4 referente à série de Fibonacci [41]:

```
1.   fib(0,1).
2.   fib(1,1).
3.   fib(M,N) :- [M1 is M-1, fib(M1, N1)]
4.                    &
5.                [M2 is M-2, fib(M2, N2)],
6.                N is N1 + N2.
```

Figura 2.4: Exemplo de programa em Prolog referente à série de Fibonacci

Observe que os dois objetivos (dentro dos colchetes) localizados nas linhas 3 e 5, não possuem quaisquer dependências de dados entre si e, portanto, podem ser executados em paralelo de forma independente (no exemplo, o operador & é utilizado para indicar a independência entre os mesmos). Apenas o último objetivo N is N1 + N2 é dependente dos resultados dos objetivos anteriores e só pode ser executado quando forem identificados os valores de N1 e N2.

O paralelismo E independente pode ser encontrado em diversas aplicações, como os algoritmos de "dividir para conquistar"(Fibonacci, quicksort, entre outros), onde um dado problema pode ser dividido em problemas menores que podem ser resolvidos de modo independente. Alguns exemplos de sistemas que exploram paralelismo E independente: &-Prolog [44, 45, 46], DASWAM [80, 81], &ACE [71, 70] e APEX [56, 57].

O **paralelismo E dependente** é a forma de paralelismo onde os objetivos dentro de uma mesma cláusula compartilham variáveis e, portanto, a execução de um pode depender do resultado de um objetivo anterior. Esse tipo de paralelismo pode ser explorado de duas maneiras: (i) seriação: os dois objetivos podem ser executados de modo independente até que uma variável compartilhada seja acessada, quando, então, a execução passará a ocorrer de forma seqüencial; (ii) sincronização (método produtor/consumidor): quando houver o compartilhamento de variáveis, um dos objetivos pertencentes a essa relação assumirá o papel de produtor em relação ao segundo, que exercerá a função de consumidor. No momento em que uma variável

compartilhada for acessada dentro do produtor, será colocada em uma estrutura (ou *stream*), a qual será usada como argumento de entrada no objetivo consumidor. Essa abordagem está presente nas Linguagens de Programação Lógica Concorrentes (*Concurrent Logic Programming Languages* ou *Commited Choice Languages*), como Parlog [21], KL1 e GHC [87] e *Concurrent Prolog* [79].

Existem diversos sistemas que exploram mais de um tipo de paralelismo, em geral, o paralelismo OU e o paralelismo E independente, como; ACE [40], PEPSys [89], ROPM [74] e AO-WAM [39]. Existem ainda os sistemas que exploram o paralelismo OU e o paralelismo E dependente, como o sistema Andorra [30, 25].

No próximo capítulo, é apresentado o sistema Aurora, sobre o qual este trabalho foi desenvolvido.

Capítulo 3

Aurora: Um Modelo de Paralelismo OU

Os computadores paralelos surgiram, no âmbito comercial, como uma possível fonte de alto poder de computação e melhor custo. Nesse momento, diversos algoritmos paralelos foram desenvolvidos e verificou-se que esta era uma tarefa bastante difícil, tornando-se um dos maiores obstáculos à disseminação da computação paralela. O que se desejava era explorar o paralelismo nas aplicações sem que, para isso, houvesse a necessidade de qualquer interferência por parte do programador. Mas, como a maioria das linguagens são baseadas em construções de von Neumann[1], que são inerentemente seqüenciais, muitas dificuldades foram encontradas ao longo desse processo.

Nesse contexto, a linguagem de programação em lógica Prolog, por ser uma das poucas linguagens não baseadas em construções von Neumann, despontou como instrumento da área para estudar o potencial dos multiprocessadores. Era apropriada para a programação de sistemas avançados e utilizada em diversas aplicações importantes, como prova de teoremas e processamento de linguagem natural.

A programação em lógica, portanto, fornecia o potencial de se explorar o paralelismo de forma implícita, retirando do programador a responsabilidade e a preocupação de gerenciar o paralelismo diretamente, ou seja, de forma explícita. Isso poderia facilitar a programação e permitir a migração de softwares de maneira transparente entre máquinas seqüenciais e paralelas.

[1]John von Neumann projetou uma máquina de propósito geral onde instruções e dados são armazenados na memória. Assim, as construções von Neumann são baseadas na estrutura tradicional dessas máquinas, onde uma CPU envia requisições seqüênciais à memória através de um barramento, que serão tratadas e respondidas também de forma seqüencial.

Para verificar a dimensão real da capacidade da programação em lógica em ambientes paralelos e determinar seus limites, foi proposto e desenvolvido o sistema Aurora, cujos propósitos eram [32]:

a) **servir de ferramenta de pesquisa de sistemas de programação lógica em paralelo:** o objetivo é avaliar os requisitos de sistemas que exploram o paralelismo em aplicações lógicas, como o Aurora; e

b) **servir de sistema de demonstração para a execução de aplicações paralelas de grande porte:** o objetivo agora é observar o comportamento dessas aplicações quando executadas em paralelo, comparando desempenho, identificando possíveis motivos de diferenças nas execuções, entre outros pontos.

O sistema Aurora é uma implementação protótipo de paralelismo OU da linguagem Prolog para multiprocessadores de memória-compartilhada, desenvolvido como parte de uma pesquisa colaborativa informal, conhecida como *Gigalips Project.* Inicialmente, o sistema Aurora foi desenvolvido para máquinas multiprocessadoras de memória-compartilhada do tipo UMA, como *Sequent Symmetry* [59] e migrado posteriormente para arquiteturas NUMA, como BBN TC-2000[2] [9]. Em uma arquitetura UMA os acessos feitos à memória são sempre realizados em tempo constante[3], independente da posição de memória solicitada. Ao contrário, em uma arquitetura NUMA (exemplo: *clusters* de máquinas de memória-compartilhada), os tempos de acesso à memória podem variar dependendo da posição que se quer acessar [41].

3.1 A Origem do Projeto Gigalips

Na Terceira Conferência Internacional em Programação em Lógica, ocorrida em Londres, no ano de mil novecentos e oitenta e seis (1986), houve um encontro de representantes de vários grupos de pesquisa interessados na área de paralelismo em programação em lógica. Desse encontro, surgiu um projeto de desenvolvimento,

[2]Arquitetura escalável com processadores Motorola 88000, que possuía memória fisicamente compartilhada e memória-distribuída, ambas suportadas em hardware.

[3]Considerando desprezível o acesso via hierarquia de memória em relação ao tempo de computação.

aberto a participação de qualquer pessoa interessada no assunto, que constantemente trocava informações e que se concentrou no desenvolvimento do sistema Aurora.

No ano seguinte, esse projeto recebeu o nome de *Gigalips Project* e seus implementadores eram grupos do *Argonne National Laboratory, University of Manchester* (que passou depois para *University of Bristol*, em 1988), *Swedish Institute of Computer Science* e *IQSOFT SZKI Intelligent Software Co. Ltd.*, em Budapeste. O *Gigalips Project* tinha o objetivo de demonstrar a eficiência da programação em lógica na exploração do paralelismo em máquinas multiprocessadoras, sendo um veículo importante para a concentração de esforços de pesquisa nesse assunto.

3.2 O Projeto

Existem diversos modelos que visam a explorar, de maneira transparente, o paralelismo da linguagem Prolog em máquinas multiprocessadoras de memória-compartilhada. Como já mencionado, Prolog oferece dois tipos principais de paralelismo: paralelismo OU e paralelismo E. O sistema Aurora explora o primeiro tipo e é baseado no modelo SRI, apresentado neste capítulo.

O paralelismo OU pode ser obtido em larga escala e granulosidade baixa em um conjunto grande de aplicações e, geralmente, se manifesta na forma de *generate and test* ou *iterate and compute*. A busca em um banco de dados dedutivo, o *parsing* de sentenças de linguagem natural, a pesquisa de *strings* em um documento ou a compilação de um conjunto de objetos são alguns exemplos de aplicação de paralelismo OU [88].

3.2.1 Os Modelos de Execução Paralela OU

Uma questão importante, na implementação do paralelismo OU, é o tratamento dado aos múltiplos ambientes que co-existem numa árvore de execução correspondente à execução de um programa. Basicamente, este problema pode ser exemplificado pelo cenário da Figura 3.1 [41].

Sejam dados dois nós em dois ramos diferentes de uma árvore-OU, que compartilham todos os nós acima (a partir) do primeiro nó ancestral comum aos dois ramos. Então, uma variável criada em um desses nós ancestrais pode assumir valores distintos em cada um dos ramos, e é desejável que os mesmos tenham escopo apenas

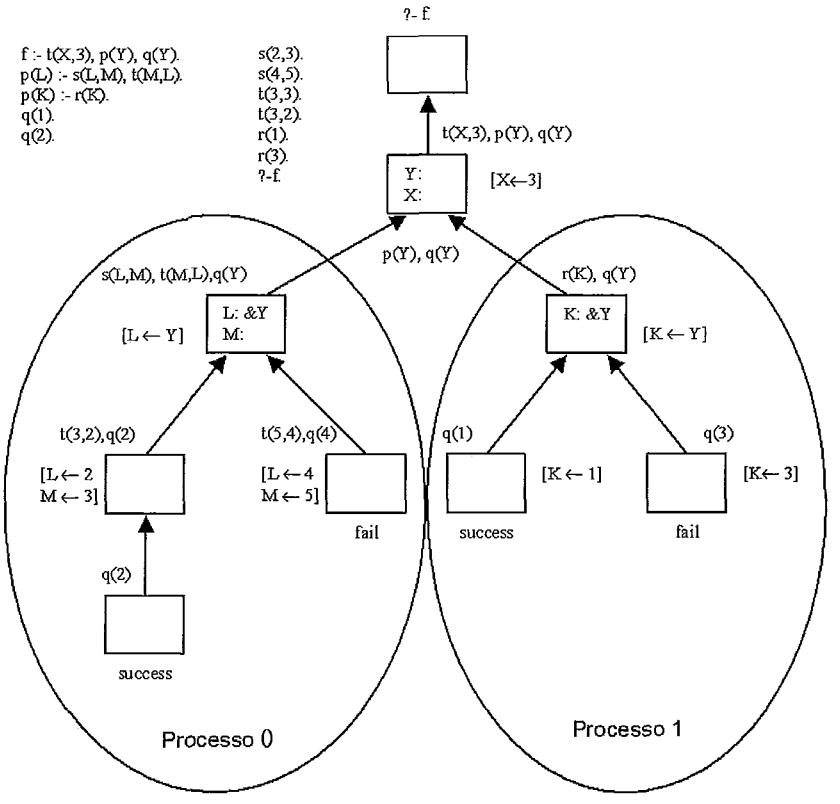


Figura 3.1: Árvore de execução do programa exemplo

no ramo a que pertencem. Isso acontece porque, na execução seqüencial, os ramos são percorridos um a um e, portanto, as variáveis vão assumindo um único valor de cada vez. Na presença de falha num ramo, todas as operações dele são desfeitas num processo denominado *backtracking*, e um próximo ramo será executado. Desse modo, uma variável antes instanciada com um valor, na operação de *backtracking* terá o mesmo desfeito e um novo poderá ser atribuído na execução do ramo seguinte.

Como na execução paralela os ramos podem ser percorridos simultaneamente, os seus respectivos ambientes devem ser organizados, de tal forma, que se possa facilmente discernir os valores aplicados à variável em cada um dos ramos.

A Figura 3.1 apresenta uma árvore-OU onde existem dois processos executando, cada qual em um ramo diferente. Observe que ambos manipulam a variável Y, atribuindo diferentes valores ou *bindings* a ela. Como esses valores são locais ao ramo e, portanto, não devem ser vistos pelos outros processos, os valores encontrados para Y não podem ser armazenados diretamente na área de memória correspondente. Do contrário, as execuções não ocorrerão como planejado. A representação desses múltiplos ambientes existentes quando exploramos o paralelismo OU está definida

nos modelos de execução e constitui o maior problema na implementação do mesmo quando o assunto é eficiência.

Os modelos que exploram o paralelismo OU variam de modelos simples a outros mais elaborados, que visam ser mais eficientes. Em GUPTA e JAYARAMAN [38] é feita uma análise de alguns modelos desse tipo. Nesse cenário, é importante manter o custo de execução de todas as operações em uma implementação paralela bem próxima à seqüencial. Para isso, é necessário avaliar questões como:

a) O custo de se criar e acessar *bindings* de variáveis: nas implementações do Prolog padrão, estas são operações muito rápidas que podem ser realizadas em tempo constante. Criar um *binding* é similar a efetuar uma escrita em memória (uma atribuição à célula de valor da variável), mais um *trailing* do endereço da variável, colocando-o numa pilha chamada *trail*. Acessar um *binding* é realizar uma leitura da memória da célula de valor da variável. É fundamental manter eficiência análoga para essas operações em uma implementação paralela OU; e

b) O custo de se criar múltiplas tarefas em um *branchpoint* (nó que não teve ainda sua execução terminada ou que seja um *fork*): este não deve ser maior do que o relativo a um *backtracking* através dessas tarefas alternativas na implementação do Prolog padrão.

Entre os modelos representantes desse tipo de paralelismo, tem-se [88]:

1. *Abstract Model of classical resolution theory* - envolve a cópia de toda estrutura herdada;

2. *Naïve Model* - neste modelo, os *bindings* são armazenados em uma lista cronológica simples;

3. *SRI Model* - originalmente proposto por David Warren no SRI, onde cada processo possui seu próprio *binding array*. É o modelo utilizado como base no sistema Aurora;

4. *Argonne Model* - projetado e implementado por Lusk e Overbeek em Argonne [16, 67], há um *hash array* gravando os *bindings* de cada aresta da árvore paralela OU;

5. *Manchester-Argone Model* - uma variante do modelo Argonne proposto por Warren, onde os *hash arrays* são criados apenas quando as arestas se tornam realmente compartilhadas;

6. *Argonne-SRI Model* - uma variante do modelo SRI o qual usa a abordagem de *favoured binding* do Modelo Argonne.

Estes modelos são apresentados no artigo [88]. Apresenta-se a seguir os modelos *Abstract Model*, base da elaboração e construção dos demais, e *SRI Model*, o qual é utilizado no sistema Aurora.

Abstract Model - O Modelo Base de Execução do Prolog Padrão

Como já visto anteriormente, a execução de uma aplicação escrita em Prolog pode ser representada através de uma estrutura em árvore, onde cada um dos nós denota uma tarefa a ser executada. Assim, o nó raiz representa a consulta inicial e os nós seguintes, as respectivas tarefas derivadas desta consulta através de resolução.

A árvore de execução é computada, no Prolog seqüencial padrão, do nó localizado no maior nível de hierarquia, e da esquerda para a direita. A cada nó percorrido, é criada uma cópia física dos objetivos (ou *goals*) herdados da tarefa oriunda do nó anterior, também denominado nó pai. Esse modelo, referenciado como *Abstract Model*, acarreta um *overhead* devido à necessidade de realizar tais cópias a cada *branchpoint*. Todavia, possui a vantagem de suportar melhor o paralelismo OU, pois cada processo (também denominado *worker*) pode trabalhar de forma totalmente independente, em dados fisicamente separados, e a implementação pode ser a mesma utilizada na seqüencial.

Há a possibilidade de uma árvore ser computada por diversos processos simultaneamente, cada qual em um ramo diferente da mesma. Quando um processo cria um nó com arcos a serem explorados à direita, possibilita que um outro selecione um desses arcos e execute esta sub-árvore. O primeiro processo a entrar em um ramo é responsável pela sua criação e o último a sair, pela sua remoção. Essas ações de montar e desmontar ramos, correspondem às técnicas de resolução e *backtracking* em Prolog.

Terminada a tarefa, o processo passa para o estado ocioso e, dessa maneira, se torna disponível para buscar outros arcos ainda não executados. Na procura de outras tarefas, a obediência ao procedimento de busca padrão do Prolog, faz com que

o processo se comporte como um processo de Prolog seqüencial, e conseqüentemente, técnicas de implementação e otimização podem ser aplicadas mais facilmente.

Além disso, é importante que as árvores de computação tenham paralelismo OU de granulosidade alta, ou seja, que suas tarefas contenham muito processamento. Assim, os processos raramente terão que trocar contexto, num método chamado *task switching ou scheduling.*

O Modelo de Execução SRI (Paralelismo OU)

O Modelo SRI foi inicialmente proposto por D.H.D. Warren no SRI em 1983, e pode ser visto como uma modificação do modelo Naïve, que acrescentou ao *Abstract Model* uma lista cronológica simples, chamada lista de *bindings* (*binding list*), para as novas variáveis, similar à *trail.* No modelo SRI, um grupo de processos, denominados *workers*, coopera na exploração de uma árvore de busca Prolog, começando no nó raiz (localizado no primeiro nível de hierarquia). Essa árvore de busca é representada por estruturas de dados bastante similares àquelas do sistema Prolog padrão, assim como às da WAM. No decorrer da execução, algumas variáveis podem tornar-se potencialmente compartilháveis através da criação de um ponto de escolha (também designado como *choicepoint*, correspondente ao *branchpoint* no *Abstract Model*). Quando isso acontece, suas células de valor não podem mais sofrer qualquer tipo de modificação, senão a execução de um processo pode acabar interferindo no comportamento e no resultado de outro (veja seção anterior).

De forma resumida, isso acontece porque cada processo pode atribuir um valor diferente a essas variáveis compartilhadas, de acordo com a necessidade de sua execução. Mas, este valor é válido apenas para esta parte da árvore sendo executada, ou seja, para esta tarefa. Outro processo, que esteja explorando outra parte da árvore, pode precisar atribuir um valor diferente a estas variáveis, de acordo com sua execução. Portanto, uma mesma variável compartilhada, pode e provavelmente assumirá diversos valores em pontos diferentes da computação. Para garantir que o valor atribuído por um processo a uma variável não afetará o valor desta em outros pontos, foi dado a cada processo um *binding array*, que pode ser considerado como uma memória local ao processo. Para ele serão copiados os *bindings* oriundos da lista de *bindings* utilizada no momento. O *binding array* contém, basicamente, a mesma informação da lista de *bindings*, armazenada de tal forma que permita o acesso ao valor de qualquer variável em tempo constante. Nele, os *bindings* são

adicionados a cada passo de resolução e removidos no *backtracking*. No *binding array* também são registrados os *bindings* condicionais, isto é, *bindings* de variáveis que foram criadas antes do último ponto de escolha. Estes *bindings* são designados como *bindings* condicionais porque dependem do resultado da execução deste ramo da árvore. Os *bindings* condicionais são também registrados em ordem cronológica na lista compartilhada de *bindings*, chamada de *trail*, similar à existente na WAM. Já os *bindings* incondicionais são implementados como na WAM, pela atualização direta na célula de valor da variável, sem a necessidade de serem colocados na *trail* ou *binding array*.

De um modo um pouco mais formal, um *binding* é dito condicional se a variável em questão é compartilhada ou potencialmente compartilhada com outro ramo da árvore, ou seja, existe um *branchpoint* entre o ponto de criação e de *binding* da variável. Do contrário o *binding* é dito incondicional.

A utilização de *binding array* e *trail* faz com que as operações básicas do Prolog de *binding*, *unbinding* e *dereferencing* sejam feitas com um mínimo de *overhead* em relação à execução seqüencial, mantendo as operações em tempo constante. Essa é uma das maiores vantagens do modelo SRI. Além disso, a maior parte das operações é bem similar a sua forma numa implementação seqüencial.

A desvantagem do modelo apresentado está no *overhead* (também chamado de *migration cost* ou *task-switching cost*) gerado pela troca de tarefa (*task switching ou scheduling*), pois o processo deve fazer com que seu *binding array* reflita a lista de *bindings* do novo nó. Embora isso possa ser feito pela reinicialização do *binding array*, normalmente o novo estado terá muito em comum com o anterior. Então, geralmente, é melhor instalar e desinstalar os *bindings*, conforme haja a necessidade de se mover na árvore de computação para buscar uma nova tarefa. Assim, os *bindings* que correspondem à parte em comum entre a lista anterior e a nova, são mantidos intactos. Resumindo, os *bindings* (da lista de *bindings*) criados antes do nó ancestral comum entre as duas tarefas são conservados no *binding array*; os posteriores são apagados e, em seu lugar, são instalados os *bindings* (da nova lista de *bindings*) criados após o nó ancestral comum. Dessa maneira, o custo de se mover entre os nós é proporcional à distância entre eles, mais precisamente o número de *bindings* encontrados no decorrer do caminho.

Na Figura 3.2, tem-se um exemplo do funcionamento dos *bindings arrays* para o exemplo de múltiplos ambientes da Figura 3.1, apresentado anteriormente.

O Modelo de Execução do Sistema Aurora

Aurora é baseado no modelo SRI, descrito anteriormente. Sua máquina Prolog usa uma versão do Sicstus Prolog [18], uma implementação portátil da WAM escrita na linguagem C, que foi modificada para suportar a execução paralela [10].

Para suportar o modelo SRI, mudanças tiveram que ser feitas no Sicstus Prolog, de forma que o mesmo pudesse implementar o mecanismo de *binding array*. Essas mudanças degradaram o desempenho seqüencial da máquina em 25% [10].

3.3 Implementação

A árvore de execução pode ser dividida em duas partes: pública e privada. A parte pública é acessível a todos os processos e corresponde à parcela superior da árvore. Já a parte privada corresponde à parcela inferior da árvore, onde cada ramo é acessível somente ao processo que nele estiver trabalhando. Essa divisão possui dois propósitos [32]:

- **Permitir que um processo em execução numa parte privada da árvore comporte-se de forma semelhante à máquina seqüencial padrão**, sem se preocupar com a sincronização ou a atualização de dados para propósitos de escalonamento; e
- **Prover um mecanismo pelo qual a granulosidade do paralelismo OU possa ser controlada.** Mantendo-se o trabalho privado, um *worker* pode prevenir que suas tarefas tornem-se muito fragmentadas.

Os sistemas paralelos em Prolog consistem, basicamente, de dois componentes: um escalonador e uma máquina prolog (*engine*):

- **Escalonador**: executado quando o *worker* está na parte pública da árvore. É responsável por encontrar e distribuir tarefas da região compartilhada para os *workers*.
- **Máquina prolog**: executada quando o processo está na parte privada da árvore. É responsável por executar o código Prolog. Periodicamente, ocorre uma pausa para que funções de escalonamento sejam efetuadas.

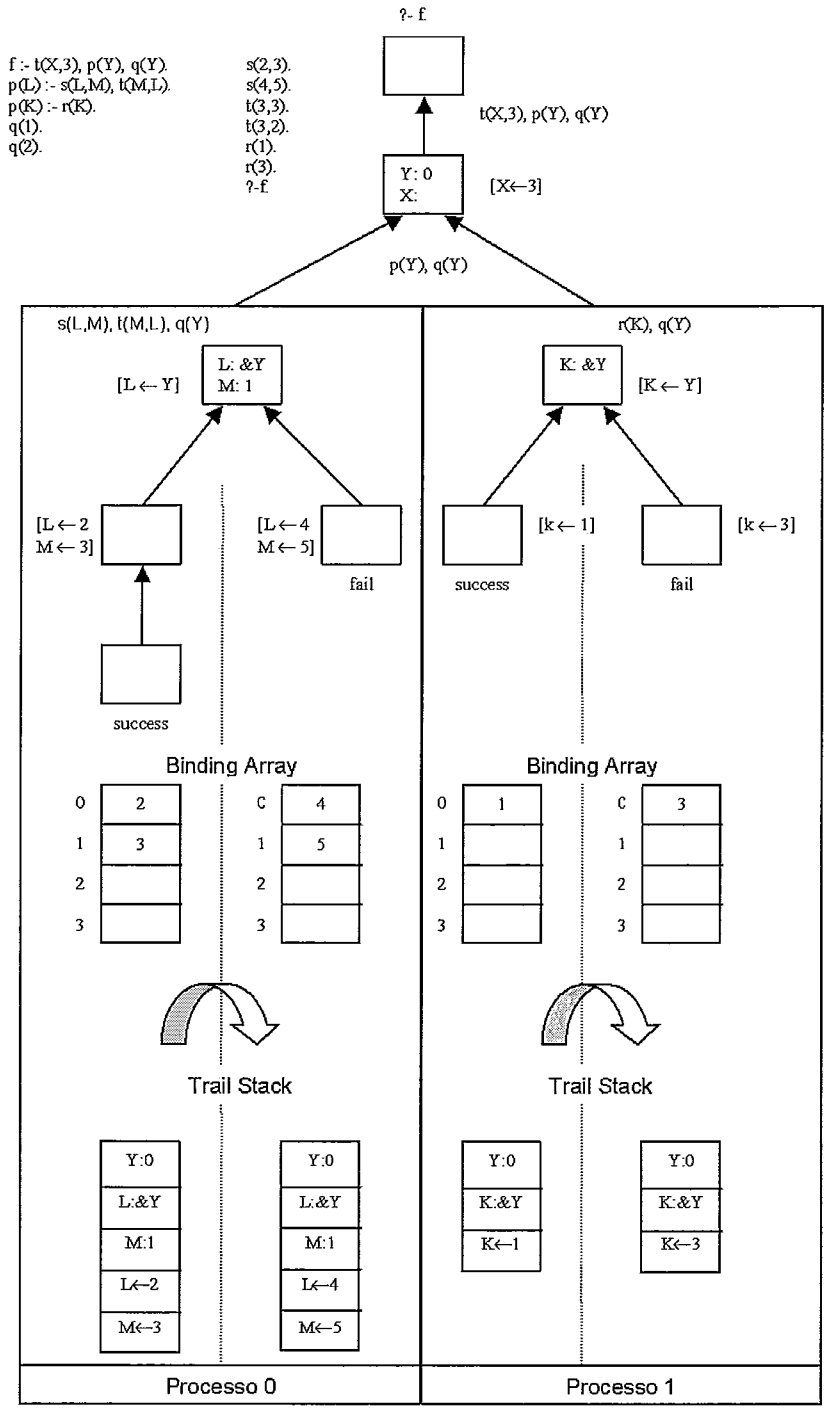


Figura 3.2: Modelo de Execução do Sistema Aurora

Esses dois componentes são separados por uma interface bem definida, que permite a combinação de diferentes máquinas prolog e escalonadores que estejam em conformidade com a mesma. Essa interface é estudada ainda neste capítulo.

Alguns escalonadores já foram utilizados com o sistema Aurora, dentre eles [10]: *Manchester Scheduler*, *Argonne Scheduler* e *Wavefront Scheduler*. Na versão deste

trabalho utiliza-se o escalonador de Bristol (*Bristol Scheduler*), apresentado a seguir.

3.3.1 Escalonador de Bristol

Basicamente, um escalonador deve realizar três (3) funções:

- Procurar por tarefa;
- Realizar a comunicação entre os *workers* para a importação ou exportação de tarefas; e
- Sincronizar os efeitos colaterais (*side-effects*), presentes em operações de leitura, escrita e corte, para manter compatibilidade com a semântica operacional seqüencial da linguagem.

Em geral, espera-se que um escalonador selecione as maiores tarefas ou aquelas que se apresentem como menos especulativas. O trabalho especulativo é aquele que pode ser desnecessário, devido a uma operação de *cut* ou *commit* em Prolog. De uma maneira simplificada, uma tarefa é considerada especulativa quando pertencente a um ramo que pode ser cancelado ao longo da execução (por exemplo, devido a uma falha). No processamento seqüencial isso não acontece porque na presença de uma falha, nenhuma das operações posteriores é executada. Como não se pode determinar antes da execução se haverá falha ou não, as tarefas que se localizam em pontos onde podem ocorrer falhas são ditas especulativas.

O problema em realizar o escalonamento é que muitas vezes esses dois requisitos nem sempre são compatíveis. Assim, as várias propostas de escalonadores implementam técnicas diferentes para a busca e alocação de tarefas aos processadores [32].

O escalonador de Bristol implementa algumas estratégias para que um *worker* ocioso busque tarefa [10]:

- *Start high*: estratégia em que o *worker* tenta encontrar trabalho no nó de menor nível de um ramo, ou seja, no nó mais alto (*topmost node*), mais próximo do nó raiz de um ramo. Visa-se minimizar a porção compartilhada (pública) da árvore de busca e, conseqüentemente, a necessidade de comunicação entre os *workers*. A desvantagem dessa estratégia está na busca por novas tarefas, uma vez que se deve fazer uma pesquisa global, provocando uma troca de

tarefa grande (*major task switch*). Essas pesquisas são custosas e podem acarretar *overheads* significativos se executadas freqüentemente, prejudicando o desempenho do sistema. Isso acontecerá se as tarefas forem pequenas e o paralelismo caracterizar-se pela granulosidade fina;

- *Start rich*: nessa estratégia, o *worker* tenta obter trabalho no final do ramo com maior número de tarefas, ou seja, o ramo mais rico da árvore. O objetivo é compartilhar o maior número de tarefas num mesmo ramo, e minimizar a necessidade de efetuar grandes trocas de tarefas pela migração para outros ramos. A desvantagem dessa estratégia é que a porção compartilhada (pública) da árvore pode se tornar muito grande. Assim, cada vez que um *worker* nessa região fizer um *backtracking*, o escalonador deverá ser invocado para sincronizar a obtenção de uma nova tarefa, provocando um aumento no número total de pequenas trocas de tarefas (*minor task switches*);

- *Start left*: estratégia projetada principalmente para uso em trabalho especulativo, para permitir que os *workers* peguem o trabalho menos especulativo de uma sub-árvore especulativa. Pela definição de *leftmost* (mais à esquerda), o trabalho mais à esquerda é também o último trabalho de um ramo. A implementação dessa estratégia requer que os *workers* tenham conhecimento sobre a ordem de disponibilidade de trabalho na árvore de busca. Isso dificulta uma implementação eficiente dessa estratégia porque precisamos evitar que haja muita sincronização, o que poderia gerar gargalos na procura por tarefa; e

- *Start random*: nessa estratégia, o trabalho é retirado do final de um ramo, sendo o mesmo selecionado de maneira aleatória.

O escalonador de Bristol tenta minimizar o *overhead* introduzido pelo escalonador, publicando seqüências de nós no lugar do procedimento padrão de se publicar os nós de forma individual. Além disso, o trabalho é retirado do *live node* mais próximo ao final de um ramo. Realiza o tratamento especial de trabalho especulativo, com o objetivo de explorar de forma eficiente o paralelismo de muitas aplicações. Na presença desse tipo de trabalho, seleciona o ramo mais à esquerda (*Start left*) que é também o menos especulativo. Do contrário, seleciona o ramo mais rico (*Start rich*), onde há a possibilidade de maior compartilhamento de tarefas. Em

alguns casos, o escalonador implementa um mecanismo de suspensão voluntária para o tratamento de trabalho especulativo. Nesse mecanismo, na ausência de trabalho obrigatório, o escalonador seleciona um trabalho especulativo e o executa por um determinado tempo. Em seguida, voluntariamente o suspende e percorre novamente a árvore, verificando se algum trabalho mandatório já foi disponibilizado [32]. Na implementação avaliada neste trabalho, não há suspensão voluntária para facilitar o trabalho de migração proposto.

3.3.2 Uma Visão Geral da Interface

A versão atual de Aurora tem sua implementação centrada na máquina prolog, que executa o código Prolog e invoca o escalonador para a realização de tarefas complementares, tais como encontrar trabalho e comunicar-se com outros *workers*, assim como para fornecer algumas informações de interesse do escalonador. A Figura 3.3 apresenta uma visão geral da atual interface entre a máquina prolog e o escalonador, utilizada no sistema Aurora. Essa interface é composta, principalmente, por funções relacionadas à procura por tarefas, funções de comunicação com outros *workers* e funções de informação para o escalonador. As mesmas são descritas a seguir:

Funções relacionadas à procura por tarefas [17]:

- `Sched_Start_Work`: é usada para obter trabalho pela primeira vez, imediatamente após a inicialização do *worker*;
- `Sched_Die_Back`: invocada quando a máquina prolog retorna para um nó público;
- `Sched_Be_Pruned`: invocada quando o *worker* tem sua tarefa cancelada por outro processo; e
- `Sched_Suspend`: invocada quando o *worker* precisa suspender seu ramo atual.

Essas funções diferem-se em suas atividades iniciais, mas normalmente têm como ação seguinte, a execução de um algoritmo comum, que procura por uma nova tarefa. Esse algoritmo tem duas saídas possíveis: ou um trabalho é encontrado, ou o sistema inteiro é terminado. De forma correspondente, como se pode observar na Figura 3.3, as funções relacionadas à procura por nova tarefa (`Sched_Start_Work`,

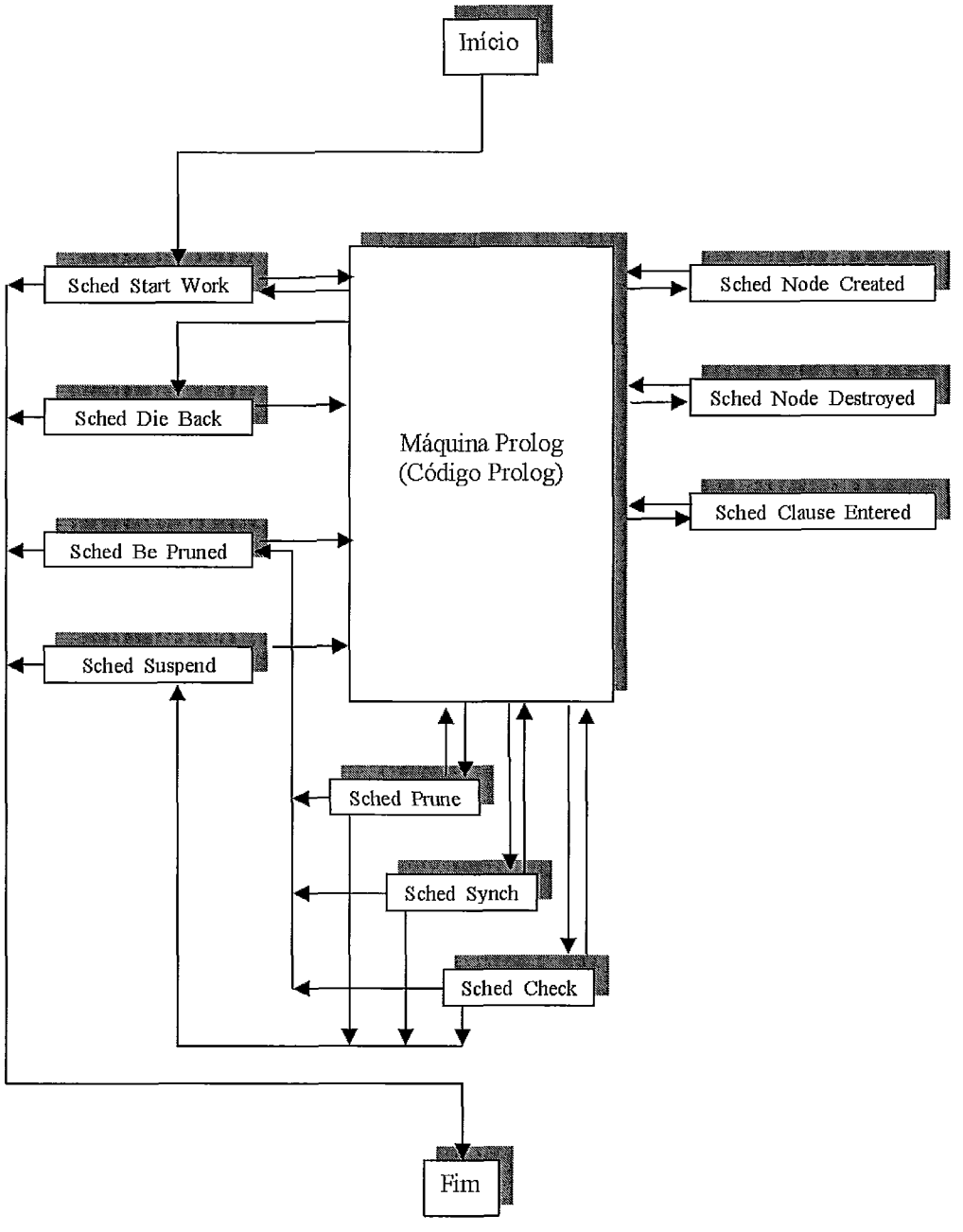


Figura 3.3: Interface provida pelo escalonador

`Sched_Die_Back`, `Sched_Be_Puned` e `Sched_Suspend`) têm duas saídas: a saída normal (setas à direita que conduzem à máquina prolog) leva o *worker* de volta à execução, enquanto a outra (setas à esquerda indicando a operação de *halt*) leva ao término do sistema.

Funções de comunicação com outros *workers*:

- `Sched_Prune`: invocada quando uma operação de *cut* ou *commit* é executada;

- `Sched_Synch`: invocada quando um predicado contendo um efeito colateral é encontrado; e

- `Sched_Check`: invocada em toda chamada de procedimento Prolog, para que o escalonador possa responder às requisições de outros *workers* sem muito atraso.

Essas funções são chamadas durante o desenvolvimento das tarefas, quando a máquina prolog pode requisitar alguma assistência do escalonador. Todas possuem três (3) saídas possíveis: uma saída normal (setas que conduzem à máquina prolog) em que o *worker* retorna ao trabalho, e mais duas, que correspondem ao término prematuro da tarefa corrente, quando o ramo atual foi cortado ou teve que suspender (setas que conduzem às operações de `Sched_Be_Pruned` e `Sched_Suspend`). Em ambos os casos, a máquina prolog realiza os procedimentos necessários para cada tipo de terminação, e procederá a chamada do escalonador através da função apropriada para encontrar trabalho.

Funções de informação para o Escalonador:

Os escalonadores podem solicitar informações sobre eventos que possam ocorrer durante a fase de execução na parte privada da árvore, tais como:

- `Sched_Node_Created`: macro invocada quando um nó é criado pelo *worker*, o que requer a atualização da árvore;

- `Sched_Node_Destroyed`: macro invocada quando um nó privado tem sua última alternativa de trabalho retirada, passando a ser identificado como um nó morto (*dead node*), ou seja, que não possui mais tarefas; e

- `Sched_Clause_Entered`: macro invocada quando a máquina prolog começa a executar uma cláusula. Objetiva manter informações sobre a presença de operadores de corte (*cuts*) que eventualmente possam se confirmar no ramo atual.

As duas primeiras funções são usadas para se ter conhecimento e controle da presença de nós paralelos na região privada - como uma fonte em perspectiva de trabalho para outros *workers*.

Outras funções do Escalonador

O escalonador também realiza outras funções, como inicialização e término do sistema e tratamento de interrupções.

Funções de inicialização e término:

No período de inicialização do sistema, a máquina prolog cria as estruturas de dados do *worker* mestre, cria o nó raiz e invoca a macro `Sched_Init`, que inicia os campos correspondentes do escalonador no nó raiz e as estruturas de dados globais necessárias ao mesmo. Em seguida, cada um dos *workers* invocará a macro `Sched_Set_Up_Worker` para criar e preencher suas estruturas de dados individuais. Como último passo nesse processo, a macro `Sched_Start_Work` deve ser executada para que os *workers* possam, de fato, começar a trabalhar. No final da computação, a macro `Sched_Deinit` deve ser chamada para liberar os espaços alocados em memória. Resumindo:

- `Sched_Init`: inicializa todos os campos específicos do escalonador no nó raiz e inicializa as estruturas de dados globais para o escalonador;
- `Sched_Set_Up_Worker`: realiza a operação de inicialização das estruturas de dados de cada *worker*, atribuindo-lhes um número de identificação para acessos posteriores; e
- `Sched_Deinit`: restaura o estado do sistema anterior à chamada de `Sched_Init`, liberando os espaços alocados em memória.

Funções de tratamento de interrupção:

O usuário do sistema Aurora pode interromper a execução e requisitar que algumas ações sejam realizadas, tais como o aborto, a saída ou mesmo a continuação da execução do sistema. Esses serviços são implementados majoritariamente na máquina prolog, mas algumas macros do escalonador também são solicitadas no tratamento dessas interrupções:

- `Sched_Block`: invocada na detecção de uma interrupção. O escalonador interromperá todos os *workers* quando os mesmos fizerem a próxima chamada a `Sched_Check`.

- `Sched_Abort`: chamada em seqüência à `Sched_Block`, quando o usuário solicita que a execução seja abortada após a interrupção.

- `Sched_Unblock`: chamada em seqüência à `Sched_Block` quando o usuário solicita que a execução não seja interrompida. Assim, a execução de todos os *workers* deve prosseguir.

3.3.3 A Interface Provida pela Máquina Prolog

Como foi dito anteriormente, a execução do sistema Aurora é centrada na máquina prolog que, entre outras ações, é responsável pela criação e manutenção das estruturas de dados do sistema, assim como pela execução do código Prolog. O escalonador é requisitado para a realização de operações na parte pública de árvore, sendo a principal delas encontrar tarefas para a máquina prolog executar. A interface do escalonador e suas respectivas funções foram estudadas na seção anterior. Nesta, são apresentadas algumas das funções que a máquina prolog fornece ao escalonador para o exercício de suas tarefas.

A máquina prolog gerencia as estruturas de dados do sistema, sendo a principal delas a estrutura `node` (que representa os nós da árvore), apresentada a seguir:

```
struct node{
    struct node *own_node;          /* physical predecessor node */
    TAGGED *trail_top;              /* top of trail stack */
    TAGGED *global_top;             /* top of global stack */
    struct alternative *next_alt;   /* alternative clause */
    struct frame *frame;            /* environment stack pointer */
    INSN *next_insn;                /* continuation */
    struct frame *local_top;        /* environment stack pointer */
    TAGGED global_var;              /* global variable number */
    TAGGED local_var;               /* local variable number */
    int level;                      /* distance from root */
    struct node *parent;            /* parent node */
    #include "sch.node.h"
    TAGGED term[];                  /* saved argument registers */
};
```

Os campos desta estrutura são divididos em duas partes: uma com informações pertinentes ao escalonador, inseridas através do arquivo `sch.node.h`, e outra à máquina prolog. Os acessos do escalonador a campos da máquina prolog (e vice-versa), são feitos através de macros providas pelo sistema, tais como:

```
int Node_Level (struct node *NODE)
```

Esta macro retorna o nível do nó (`NODE`) na árvore de execução, que é preenchido pela máquina prolog em todos os nós.

```
struct node *Node_Parent (struct node *NODE)
```

Esta macro retorna um ponteiro para o nó pai do nó atual (`NODE`), informação que também é preenchida pela máquina prolog em todos os nós.

```
struct alternative *Node_Alternatives (struct node *NODE)
```

De forma similar às anteriores, esta macro retorna o conteúdo do campo `next alternative` do nó (`NODE`), que indica qual a próxima alternativa a ser executada.

A máquina prolog também descreve algumas outras macros para a realização de tarefas distintas, das quais algumas são descritas a seguir.

Funções de Notificação de Trabalho Encontrado:

```
void Found_Resumed_Work (struct node * NEW_SENTRY)
```

Esta macro notifica a máquina prolog de que um ramo suspenso (`NEW_SENTRY`) será o próximo trabalho designado para o nó.

```
void Found_New_Work(struct node *NEW_SENTRY, struct alternative *ALT)
```

Esta macro informa à máquina prolog a próxima tarefa corresponde a uma nova alternativa reservada e ainda não explorada. `NEW_SENTRY` representa o nó alocado usando a macro `Allocate_Node` e `ALT` à alternativa reservada no nó pai do primeiro.

Funções de Movimentação na Árvore:

```
void Move_Engine_Down(struct node *DOWN_TO)
```

Esta macro atualiza as estruturas de dados específicas da máquina prolog (normalmente apenas o *binding array*) para o movimento da posição corrente para o nó `DOWN_TO` (atualizando a indicação de posição atual para este nó).

```
void Move_Engine_Up(struct node *UP_TO)
```

Esta macro atualiza as estruturas de dados específicas da máquina prolog para o movimento da posição corrente para o nó `UP_TO` (atualizando a indicação de posição atual para este nó).

Função de Alocação de Nós:

```
void Allocate_Node(struct node *PARENT, struct node *EMBRYONIC)
```

Macro que aloca um novo nó, inicia os campos referentes ao nível e ao pai deste, e retorna um ponteiro para o mesmo no argumento `EMBRYONIC`.

Funções de Recuperação de Nós:

```
void Mark_Node_Reclaimable(struct node *NODE)
```

Macro que permite ao escalonador avisar a máquina prolog de que o nó (`NODE`) não é mais necessário para a computação.

```
void Mark_suspended_Branch_Reclaimable(struct node *SENTRY)
```

Esta macro deve ser usada para apagar um ramo (indicado por SENTRY) que foi suspenso e posteriormente cortado.

Função de Extensão de Região Pública:

```
void Make_Public(struct node *NEW_SENTRY)
```

Esta macro deve ser invocada quando o escalonador vai estender a região pública da árvore para o nó pai de `NEW_SENTRY`. A máquina prolog tem então a oportunidade de fazer quaisquer inicializações nos nós que irão se tornar públicos.

Capítulo 4

Processo de Adaptação do Sistema Aurora ao Software DSM TreadMarks

Neste capítulo são apresentados os principais problemas encontrados no processo de adaptação do sistema Aurora ao software DSM TreadMarks numa plataforma Linux. O objetivo é avaliar a complexidade de se portar um sistema de larga escala, baseado no modelo de memória-compartilhada, para um ambiente de memória-compartilhada distribuída. Nesse processo, a escolha do sistema Aurora, apresentado no Capítulo 3, baseou-se na sua relevância e aplicabilidade na área de Inteligência Artificial como: (a) ferramenta de pesquisa de sistemas de programação lógica em paralelo e (b) sistema de demonstração para a execução de aplicações paralelas de grande porte. Infelizmente, o custo elevado das arquiteturas de memória-compartilhada tem restringido e, muitas vezes, impossibilitado o seu uso, assim como o de diversos outros sistemas desse tipo. Esse problema constituiu a principal motivação para o desenvolvimento desse trabalho. Já o software DSM TreadMarks foi escolhido por ter se firmado no meio acadêmico como a ferramenta mais utilizada e melhor elaborada para o objetivo a que se propõe, ou seja, de constituir uma interface entre sistemas projetados para memória-compartilhada e arquiteturas de memória-distribuída.

4.1 Etapas da Adaptação do Sistema Aurora ao Software DSM TreadMarks

Para realizar a adaptação do sistema Aurora ao software DSM TreadMarks diversas etapas foram cumpridas, dentre as quais são apresentadas, a seguir, as mais importantes:

4.1.1 Iniciação do ambiente TreadMarks e criação dos processos escravos

A iniciação da biblioteca e do ambiente de TreadMarks, e a criação de processos escravos é feita por meio da chamada à função `Tmk_startup(argc,argv)` que deve ser obrigatoriamente a primeira rotina de TreadMarks a ser executada numa aplicação. Depois de `Tmk_startup(argc,argv)`, as memórias privada e compartilhada são idênticas para todos os processos, com exceção da variável global `Tmk_proc_id`, identificação individual de cada processo.

```
#ifdef Tmk_linux
  printf("Aurora DSM com TreadMarks numa Plataforma Linux.\n");
  Tmk_startup(argc, argv);
  printf("Tmk_startup realizado com sucesso!\n");
  number_of_workers = Tmk_nprocs;
  printf("Tmk_proc_id: %d\n", Tmk_proc_id);
#else

#if defined(__svr4__) || defined(__linux__)
    /* vsc: we're using mcc's library */

#ifdef SBA
#ifdef __linux__
  shm_init( SBA_SIZE );
#else
  shm_init( 4*1024*1024, shm_access_type );
#endif __linux__
#else SBA
  shm_init( 12*1024*1024);      /* Alocação de memória-compartilhada */
#endif SBA

#endif

#endif Tmk_linux
```

Figura 4.1: Código de iniciação

4.1.2 Alocação de Memória-compartilhada

No sistema Aurora original, a criação de memória-compartilhada é realizada de uma única vez, através da chamada à função `shm_init(size)`, apresentada na Figura 4.1. Esta aloca uma grande área de memória-compartilhada (do tamanho de

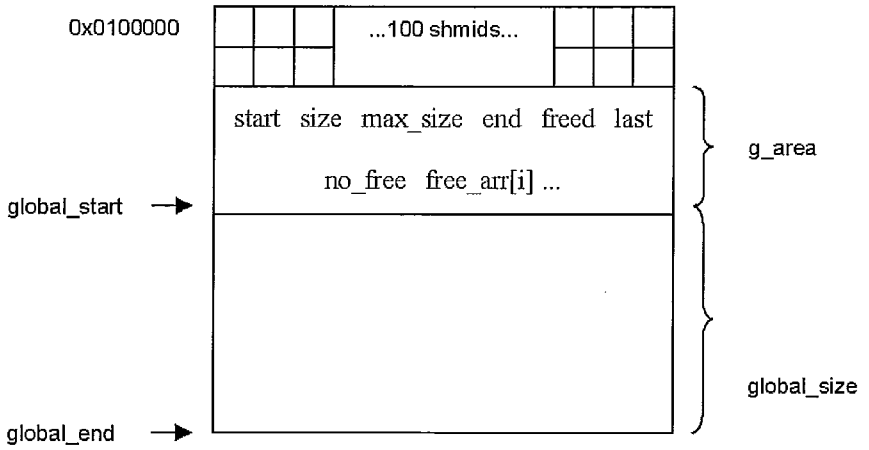
size) inicialmente vazia, que vai sendo progressivamente preenchida, conforme haja necessidade de se compartilhar dados entre os processos. A memória-compartilhada é representada por um conjunto de blocos, denominados *chunks*, que podem ser representados de forma simplificada como na Figura 4.2.

O sistema disponibiliza um conjunto de `100 shmids`, onde cada qual contém a identificação de um *chunk* de memória utilizado. No início da execução de Aurora, o sistema aloca um tamanho de memória igual a `12*1024*1024 bytes`. Como, na versão para Linux, um *chunk* pode ter até `16*1024*1024 bytes`, apenas um (1) bloco é suficiente e necessário para que se dê início ao compartilhamento de dados, sendo o mesmo referenciado por `shmid[0]`. Os demais, até este momento, não são utilizados.

A estrutura `g_area`, mostrada na Figura 4.2, conserva informações sobre o *chunk* atual. Observando a Figura 4.2, vê-se que nela há referência para o início (`start`) e fim (`end`) da área compartilhada ainda não alocada, e informações como o tamanho atual disponível(`size`) e tamanho máximo (`max_size`) que se pode encontrar no bloco para alocação. A estrutura `free_arr[i]` é empregada na gerência dos espaços livres em memória, onde pode auxiliar na busca pelo espaço mais adequado para um tamanho requisitado de memória. Seus índices estão relacionados com o tamanho das áreas livres, como se pode conferir na Tabela 4.1:

Índice	Tamanho mínimo (Bytes)	Tamanho máximo (Bytes)
00	1	31
01	32 (2**5)	47
02	48	63
03	64 (2**6)	79
04	80	95
05	96	111
06	112	127
07	128 (2**7)	255
08	256 (2**8)	511
09	512 (2**9)	1023
10	1024 (2**10)	2047
11	2048 (2**11)	4095
12	4096 (2**12)	8191
13	8192 (2**13)	16383
14	16384 (2**14)	32767
15	32768 (2**15)	2**24

Tabela 4.1: Gerenciamento de memória livre



shmids: ids dos chunks de área compartilhada
g_area: estrutura de dados que guarda informações sobre o chunk atual, tais como:
start: ponteiro para o início da área compartilhada disponível no chunk;
size: tamanho da área compartilhada disponível no chunk;
max_size: tamanho máximo da área compartilhada;
end: ponteiro para o final da área compartilhada;
freed: contador das regiões liberadas na memória-compartilhada;
last: índice para a última região liberada na memória-compartilhada;
no_free: quantidade de regiões ocupadas na memória-compartilhada;
free_arr[i]: vetor cujos índices estão associados ao tamanho das regiões livres. Funciona como um ponteiro para uma lista de regiões livres da memória-compartilhada.
global_start: ponteiro temporário para o preenchimento das informações no chunk.
global_size: tamanho da área de memória-compartilhada disponível para alocação no chunk.
global_end: ponteiro para o final da área de memória-compartilhada do chunk atual.

Figura 4.2: Processo de alocação de memória-compartilhada

Os blocos também possuem ponteiros temporários, como `global_start`, `global_end` e `global_size` que, respectivamente, indicam as posições de início e fim da área compartilhada de um bloco, e seu tamanho.

Após a criação da memória-compartilhada, a alocação dos dados nessas áreas é feita através de chamadas sucessivas à função `shmalloc()`, que utilizará `free_arr[i]` para encontrar no bloco o espaço livre mais próximo do tamanho desejado. Pode-se observar, portanto, que o processo de criação de memória-compartilhada é distinto e independente do processo de alocação dos dados (variáveis ou estruturas) na mesma, embora funcionalmente estejam relacionados.

Na versão de Aurora com TreadMarks, o mesmo não acontece, pois o software DSM já faz todo o gerenciamento de memória. Nele, o processo de criação de memória-compartilhada está associado, de forma intrínseca, a uma variável ou estrutura. Assim, no início de sua execução, o sistema invoca a rotina `Tmk_startup(argc,argv)`, apresentada na seção anterior, que iguala as memórias

de todos os processos, estabelecendo um endereço virtual (0x50000000) que é usado como base de todas as alocações em memória-compartilhada. Conforme sejam inseridas na memória, as variáveis vão recebendo endereços virtuais consecutivos, fornecendo a visão lógica de uma memória-compartilhada com alocações contíguas, como pode ser visto na Figura 4.3. Mas, na verdade, a memória-compartilhada é alocada aos poucos durante a execução, conforme seja necessário colocar dados nessa área, por meio da chamada à função `Tmk_malloc(size)`.

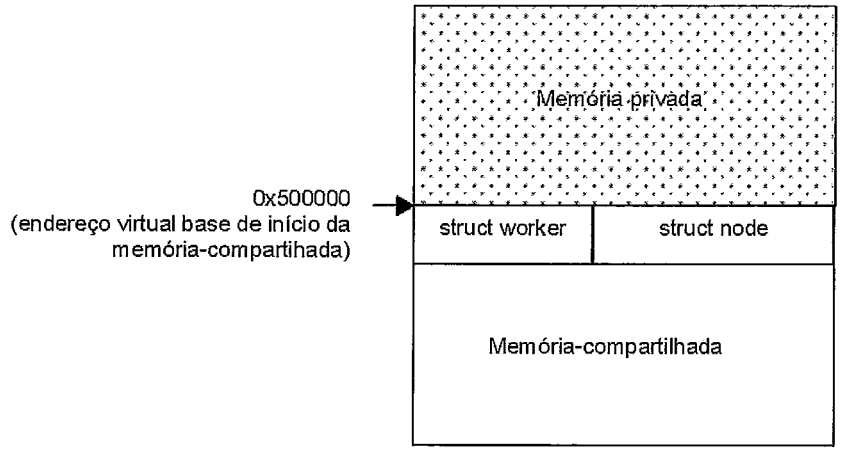


Figura 4.3: Processo de alocação de memória no Software DSM TreadMarks

É importante observar, entretanto, que alocar um dado na memória-compartilhada não significa que o mesmo possa ser visto por todos os processos. Para isso, é preciso entregar-lhes o endereço da variável na memória-compartilhada e o endereço da variável local onde o mesmo deve ser guardado. Esse procedimento é feito pela rotina `Tmk_distribute(ptr, size)` do TreadMarks. O trecho de código da Figura 4.4 exemplifica o uso dessas funções.

```
void * p = (void *)Tmk_malloc(size);
Tmk_distribute(&p, sizeof(p));
```

Figura 4.4: Exemplo de utilização de `Tmk_distribute`

A variável privada escolhida para conter o endereço da memória-compartilhada deve ser global, de modo que todos os processos utilizem as mesmas variáveis para referenciar os dados que estejam em memória-compartilhada. No exemplo da Figura 4.4, imagine que `p` esteja no endereço privado `0x22500000` e receba como retorno da função `Tmk_malloc(size)` a referência para a nova área de memória-compartilhada que começa no endereço `0x51000000`. Então, no momento

de distribuir essa informação para os demais processos, deve-se enviar: (a) o seu endereço privado para que todos os processos saibam onde colocar a nova informação, e (b) o endereço da nova área de memória-compartilhada, que é referenciada por p. Desse modo, para que todos os processos tenham as mesmas variáveis e possam compartilhar memória, é importante que as áreas de código e dados sejam iguais para todos os processos do sistema. O procedimento de criar os processos escravos e igualar suas memórias é feito durante a execução de `Tmk_startup(size)`.

A Figura 4.5 apresenta o esquema de memória para os processos após a distribuição de p:

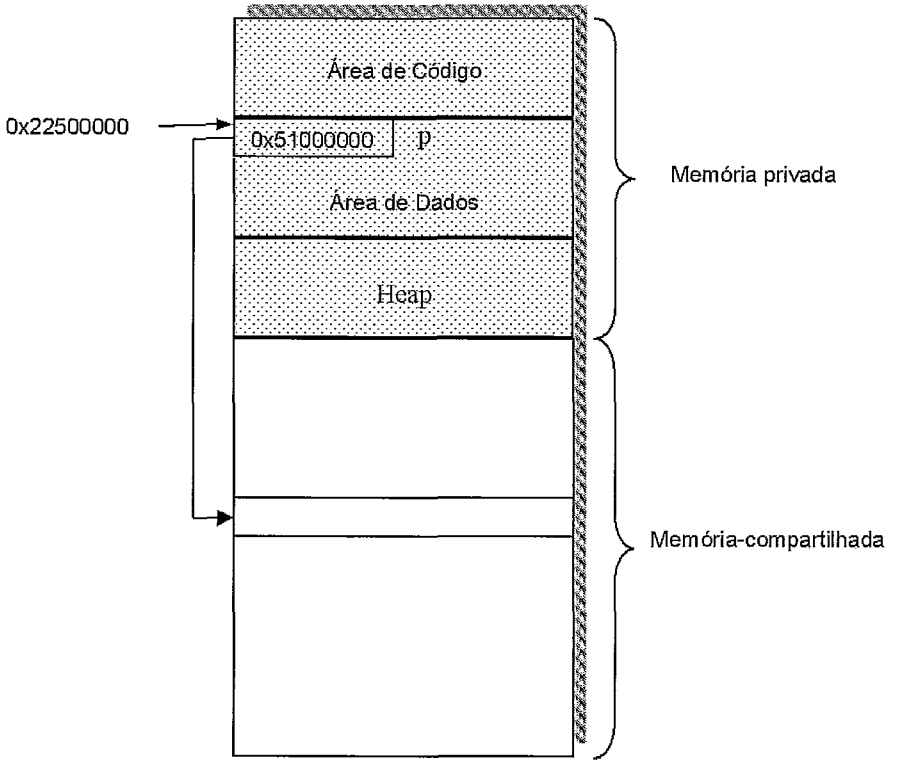


Figura 4.5: Esquema de memória de um processo após a operação `Tmk_distribute`

Portanto, o TreadMarks tem a responsabilidade de criar e manter as áreas compartilhadas do sistema, realizando todo o processo de gerência de memória, antes sob o encargo do próprio sistema.

Como se pode observar, o procedimento de criação de memória-compartilhada nas versões original e DSM de Aurora são bastante diferentes. Na tese, para introduzir o esquema de memória utilizado pelo TreadMarks, foi preciso fazer um estudo meticuloso sobre o funcionamento da gerência de memória do sistema

para arquiteturas compartilhadas. Determinou-se todas as funções relacionadas, suas funcionalidades e a forma de substituí-las pelas rotinas correspondentes de TreadMarks, o que em princípio parecia uma tarefa relativamente simples, mas que se revelou complexa e exigiu tempo.

Um segundo problema enfrentado nessa fase foi a forma como o sistema estava estruturado, o que conflitava com o modo do TreadMarks operar. No início da execução da versão original de Aurora, existe apenas um processo, denominado mestre. Este faz a criação de todo o ambiente necessário para a aplicação executar suas tarefas, inclusive realizando a gerência de memória privada e compartilhada. Quando toda a configuração inicial termina, os processos escravos são criados através da chamada de sistema `fork()`, do linux. Na versão TreadMarks, tentou-se manter essa estrutura. Mas, como o processo mestre precisa criar e gerenciar diversas áreas de memória-compartilhada, houve a necessidade de se fazer a primeira mudança expressiva no comportamento do sistema. Isso é decorrente do fato de que, em TreadMarks, não pode haver alocação de memória-compartilhada (`Tmk_malloc(size)`) antes da configuração do ambiente do software e da criação dos processos escravos (`Tmk_startup()`). Como não havia outra solução, os processos escravos passaram a ser criados no início da execução e esperavam o mestre realizar as configurações necessárias para que os mesmos pudessem começar a trabalhar. Mas, isso não foi suficiente. Observou-se que diversas partes da memória dos processos escravos não tinham sido iniciadas.

Na versão original, o processo mestre cria a maior parte do ambiente de memória-compartilhada e privada. No momento da chamada `fork()`, toda a área de dados e código é, então, copiada dele para os processos escravos. Dessa forma, diversos dados privados já inicializados, são herdados e utilizados ao longo da execução. Na versão DSM, como os processos escravos são criados antes dos dados serem incluídos na memória, não herdam os dados privados inicializados pelo mestre, apenas os compartilhados, que são distribuídos através de `Tmk_distribute(&p, sizeof(p))`.

A solução para esse problema foi permitir que cada processo fizesse a criação de seus dados privados, deixando ao encargo do mestre a criação das variáveis compartilhadas. Esses dados privados podem ser:

a) informações pertinentes ao processo. Nesse caso, as escritas feitas nessas variáveis não precisam ser divulgadas a outros processos; e

b) informações pertinentes a todos os processos. Funcionalmente, são variáveis compartilhadas com acessos apenas de leitura, ou seja, variáveis escritas apenas uma vez, durante sua criação, e depois compartilhadas para leitura. Na versão original, correspondem, por exemplo, às variáveis da máquina prolog, que são criadas pelo mestre em memória privada e copiadas pela chamada `fork()` aos processos escravos. A partir de então, são usadas apenas para acessos de leitura.

A versão original de Aurora permite a alocação de memória-compartilhada que não seja vista por todos os processos, ou seja, o ponteiro que a referencia é guardado em variável local. Na versão DSM, esse caso foi adaptado de modo semelhante. A área de memória-compartilhada é alocada com `Tmk_malloc(size)`, mas não é distribuída para os outros processos, ou seja, não se faz a chamada a `Tmk_distribute(&p, sizeof(p))`.

4.1.3 Sincronização das Variáveis

De um modo geral, em sistemas onde há compartilhamento de dados, deve existir um mecanismo de sincronização para todas as variáveis que: (a) sejam acessadas por dois ou mais processos e (b) tenham pelo menos uma operação de escrita.

Na versão para arquiteturas compartilhadas, o sistema Aurora provê primitivas de sincronização para controlar o acesso de escrita dos processos, de forma que uma mesma variável não seja atualizada por mais de um processo simultaneamente. Assim, a sincronização é feita somente na operação de escrita das variáveis de múltiplos escritores; a leitura destas e as variáveis de apenas um escritor não são protegidas, como seria esperado. Pode acontecer, por conseguinte, de uma variável ser escrita e lida de modo concomitante, o que poderia gerar valores incorretos e, assim, provocar erros nos resultados da execução. No entanto, isto não é um problema em Aurora, porque todas as leituras, de dados compartilhados, estão em seções de código que não precisam de seus valores mais recentes, de forma imediata e atômica, para executar corretamente.

O TreadMarks provê as primitivas de sincronização *lock* e *unlock* para o controle do acesso às regiões críticas do sistema e para a atualização da memória-compartilhada distribuída. Todavia, é responsabilidade do usuário determinar quais variáveis ou regiões necessitam de sincronização, de modo que

as primitivas permitam a execução correta da aplicação. É preciso, portanto, que se tenha um bom conhecimento sobre o sistema a ser adaptado para memória DSM.

Em geral, no TreadMarks todas as variáveis acessadas por dois ou mais processos com pelo menos uma operação de escrita, devem ser sincronizadas. Do mesmo modo, todos os processos com variáveis compartilhadas somente para leitura, devem sincronizá-las pelo menos uma vez para obter o seu conteúdo. Assim, para adaptar o sistema Aurora ao TreadMarks, houve a necessidade de se fazer um levantamento de todas as variáveis compartilhadas, de forma a verificar quais deveriam ser sincronizadas. Os casos encontrados no sistema e algumas variáveis representativas que necessitaram de sincronização foram:

a) **Variáveis de um leitor e um escritor:** `work_given (struct worker)`, `work_node (struct worker)`, `work_wanted_by (struct_worker)`;

b) **Variáveis de um leitor e múltiplos escritores:** `interested_workers_count (struct node)`;

c) **Variáveis de múltiplos leitores e um escritor:** `richness (struct worker)`, `has_nonspec_work (struct_worker)`, `glob_ready`;

d) **Variáveis de múltiplos leitores e múltiplos escritores:** `interrupt_flags (struct worker)`, `richness (struct node)`, `node_flags (struct node)`.

Em todos os casos, é importante que, na versão DSM, a sincronização seja feita não somente na escrita, como também na leitura das variáveis, porque é através da operação de sincronização por *lock* que um processo consegue acesso aos dados atualizados.

Pode-se dizer que, no TreadMarks, a operação de *lock/unlock* encapsula duas funcionalidades principais:

a) Proteção das regiões críticas, impedindo que as variáveis sejam escritas ou lidas por mais de um processo ao mesmo tempo; e

b) Atualização da memória-compartilhada distribuída. Como na versão original a memória é fisicamente compartilhada, excetuando-se o problema da leitura e escrita simultâneas nas variáveis, o seu conteúdo é facilmente acessado por

todos os processos. O mesmo não ocorre na versão DSM, uma vez que a memória é fisicamente distribuída. Como foi visto no Capítulo 2, para se obter o valor atualizado de uma variável, primeiro é preciso efetuar a operação de *lock* para obter a informação de quais páginas foram atualizadas em outros nós. Na primeira tentativa de acesso a uma variável dessas páginas, ocorrerá um *page fault*. Nesse momento, os *diffs* (dados atualizados) são, então, trazidos para a memória local.

Houve, também, o caso apresentado anteriormente das variáveis compartilhadas somente para leitura, mas colocadas em memória privada pelo processo mestre na inicialização do sistema. A distribuição delas ocorria na criação dos processos escravos, através da chamada ao sistema `fork()`, quando toda a memória do mestre era copiada para os demais processos. Como pôde ser visto, nesse tipo de variável optou-se por mantê-las em memória privada e designar a cada processo a responsabilidade pela criação de sua cópia.

A necessidade de se introduzir primitivas de sincronização para leitura e escrita de todas as regiões críticas do sistema, acarreta um tráfego excessivo na rede para atualização da memória e atraso na execução. Para diminuir esse problema, a operação de *lock* pode ser dividida em duas operações distintas: *lock* de leitura (`lock_read`) e *lock* de escrita (`lock_write`). Em sua implementação, um *lock* de leitura permitiria o acesso simultâneo de múltiplos processos a regiões críticas, para as quais não houvesse um *lock* correspondente de escrita. Uma otimização seria impedir que o processo detentor deste *lock* tentasse efetuar uma operação de escrita. Do mesmo modo, antes de liberar o acesso à região crítica solicitada, um *lock* de escrita teria que: (a) bloquear os próximos pedidos de *locks* de leitura para esta região e (b) aguardar até que todos os processos, ainda em leitura, saíssem de sua região crítica.

Implementação da Sincronização do Sistema

A sincronização do sistema Aurora, para arquiteturas DSM, utilizou as duas primitivas de sincronização oferecidas pelo TreadMarks, *locks* e barreiras. A implementação da primitiva de *lock* foi baseada numa implementação anterior de HUANG *et al.* [47] para o Andorra [25], um sistema que explora os paralelismos E e OU de programas em lógica.

Barreiras

Uma barreira impede o progresso de um processo até que todos os outros alcancem o ponto de execução desejado. O número de objetos desse tipo disponível ao usuário, é definido por `TMK_NBARRIERS`, onde cada barreira é identificada por um número inteiro no intervalo:

$[0 \leq \texttt{x} < \texttt{TMK\_NBARRIERS} - 1]$

No Aurora, as barreiras foram utilizadas, basicamente, no processo de inicialização e término do sistema. No primeiro, os processos escravos precisam esperar o mestre criar todo o ambiente de execução e obter informações das áreas de memória-compartilhada criadas e manipuladas. No segundo, o processo mestre espera até que todos os escravos terminem sua computação para, então, liberar as áreas alocadas e encerrar o programa.

Locks

Inicialmente, considere que regiões críticas ou variáveis compartilhadas do sistema devem estar delimitadas por primitivas de `lock(x)` e `unlock(x)`, como na Figura 4.6.

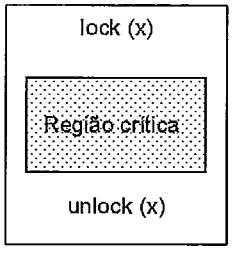


Figura 4.6: Delimitação de regiões críticas

Onde `x` é um número inteiro, dentro do intervalo de *locks* permitido pelo TreadMarks e definido em `TMK_NLOCKS`. Então:

$TMK\_NLOCKS = 1024 \leftarrow 0 \leq x < 1024$

Na versão original de Aurora, foram definidas variáveis de *lock*, sobre as quais incidiam as operações de sincronização. Assim, quando se queria escrever, por exemplo, na variável compartilhada `interested_workers_count`, invocava-se `lock(x)`, passando como argumento a sua variável de *lock* correspondente, `interest_lock`. Esta era atualizada de forma atômica, passando a conter o valor `1`, que indicava aos outros processos que a variável `interested_workers_count` estava atribuída a algum outro processo. Já na versão DSM, como visto anteriormente,

as operações de *lock* devem receber números inteiros como parâmetro, dentro do intervalo definido por `TMK_NLOCKS`. Logo, o próximo passo é definir um método para associar as variáveis compartilhadas a números inteiros. Nesse trabalho, utilizou-se a função proposta por HUANG *et al.* [47]:

```
#define LOCK_RANGE 1024
unsigned int x = (unsigned int) &(var_lck) % (LOCK_RANGE) + 1
```

LOCK_RANGE contém o valor de TMK_NLOCKS e `var_lck` é a variável de Aurora que se quer proteger.

No exemplo dado, ao se desejar escrever na variável do sistema `interested_workers_count`, deve-se substituir `var_lck` pela sua variável correspondente de *lock*, `interest_lock`.

Pode ocorrer, entretanto, de duas ou mais variáveis obterem o mesmo valor para `x`. Nesse caso, ainda que não estejam relacionadas, seus acessos são sincronizados, o que não é desejável para o sistema. Além disso, essa função não permite que um processo faça operações consecutivas de `lock(x)` num mesmo número, para o qual ainda não realizou a operação de `unlock(x)`. Então, se duas variáveis de um processo são mapeadas para o mesmo número de *lock*, são sincronizadas, ou seja, apenas uma delas pode ingressar em sua região crítica em um determinado momento. E, se o processo tentar entrar na região crítica de ambas, fará um `lock (x)` duas vezes consecutivas, sem a operação de `unlock(x)` correspondente. Quando o software DSM detectar essa tentativa, ocorre um *reacquired lock*, indicando a tentativa de readquirir um número de *lock*, gerando falha na execução.

A solução encontrada para esse problema, foi modificar a forma como a sincronização é feita. Portanto, foram redefinidas as variáveis de *lock* para ajudar no controle de acesso às regiões críticas.

```
typedef struct {
  unsigned proc_id;      /* Identificação do processo */
  unsigned numlocks;     /* Indica o número de locks */
                         /* consecutivos que o processo executou */
} slock_t;

typedef slock_t LOCKTYPE;
```

Assim, antes de entrar numa região crítica, o processo deve invocar a macro `LOCK(var_lck)`, apresentada na Figura 4.7 e realizar as seguintes tarefas:

1. Verificar se a variável já está atribuída (*lock*) a ele. Nesse caso, `numlocks` é incrementado, contendo o número de operações de `LOCK` que um mesmo processo efetuou, sem ainda ter sido feita a operação correspondente de `UNLOCK`. Esse passo evita que no programa ocorra um erro devido à tentativa de um processo requisitar uma variável pela chamada a `Tmk_lock_acquire (x)` sem antes libera-la, ou seja, sem invocar `Tmk_lock_release(x)`;

   `proc_id = pid do processo (getpid()*100 + Tmk_proc_id)` → variável já está atribuída; `numlocks++`; sair;

2. Calcular o número do *lock* (valor de `x`) que será associado à variável de *lock*, no caso do passo anterior não ser verdade;

3. Fazer uma chamada à operação de *lock* do TreadMarks, denominada `Tmk_lock_acquire (x)`.

4. Verificar o valor do campo `proc_id`.

   `proc_id = 0` → variável liberada; `proc_id = getpid()*100 + Tmk_proc_id`; `numlocks++`; o processo pode entrar na região crítica.

   `proc_id != 0` → variável atribuída a outro processo; esperar; e

5. Fazer uma chamada à operação de *unlock* do TreadMarks, denominada `Tmk_lock_release (x)`.

De modo similar, a chamada à macro `UNLOCK(X)`, mostrada na Figura 4.8, deve realizar as seguintes tarefas:

1. Calcular o número do *lock* (valor de `x`) que será associado à variável de *lock*;

2. Fazer uma chamada a `Tmk_lock_acquire(x)`;

3. Decrementar `numlocks` e verifica se o mesmo chegou a zero. Em caso afirmativo: `proc_id = 0`; e

4. Realizar a chamada à operação de *unlock*, `Tmk_lock_release (x)`.

```
#define LOCK(var_lck) \ { \
  int lck_count = 0;\
  unsigned int x = (unsigned int) &(var_lck) % (LOCK_RANGE) + 1;\
  while (1){ \
    if ((var_lck).proc_id == getpid()*100 + Tmk_proc_id){\
      (var_lck).numlocks++;\
      break;\
    }\
    else{\
      Tmk_lock_acquire(x); \
      if ((var_lck).proc_id == 0) {\
        (var_lck).proc_id = getpid()*100 + Tmk_proc_id;\
        (var_lck).numlocks++;\
        Tmk_lock_release(x); \
        break;\
      }\
      else{\
        Tmk_lock_release(x); \
        lck_count++;\
        if (lck_count > MAX_LOCK_TRIES) {\
           printf("DeadLock\n");\
           exit(1);\
        } else if (lck_count % 10 == 0 || lck_count >
           (MAX_LOCK_TRIES - 10)) {\
           printf("Esperando liberar lock \n");\
        }\
        TMK_SLEEP(1);\
      }\
    }\
  }\
}
```

Figura 4.7: Macro LOCK

Como se pode observar, os campos da variável de *lock* informam o estado da região crítica, ou seja, se a mesma está ocupada ou liberada. Sendo assim, como se pode ver na Figura 4.9, uma operação de *Tmk_lock_acquire/Tmk_lock_release* protege apenas a variável de *lock*, para que sejam feitas as leituras e/ou modificações necessárias em seus campos.

Embora, a variável de *lock* do processo não esteja atribuída a ele quando o mesmo ingressar na região crítica correspondente, suas escritas estarão protegidas, pois em todos os outros processos, `proc_id` estará diferente de zero, indicando que a região

```
#define UNLOCK(var_lck)\ { \
  unsigned int x = (unsigned int) &(var_lck) % (LOCK_RANGE) + 1
  Tmk_lock_acquire(x);\
  if ((var_lck).proc_id == getpid()*100 + Tmk_proc_id){\
    (var_lck).numlocks--;\
    if((var_lck).numlocks == 0){\
      (var_lck).proc_id = 0;\
    }\
    else if((var_lck).numlocks < 0){\
      Tmk_lock_release(x);\
      exit(1);\
    }\
  }\
  else{\
    Tmk_lock_release(x);\
    printf("Unlock inválido\n");\
    exit(1);\
  }\
  Tmk_lock_release(x);\
}
```

Figura 4.8: Macro UNLOCK

está ocupada.

Com essa solução, variáveis independentes associadas ao mesmo *lock*, poderão ser acessadas sem que uma impeça a progressão da outra.

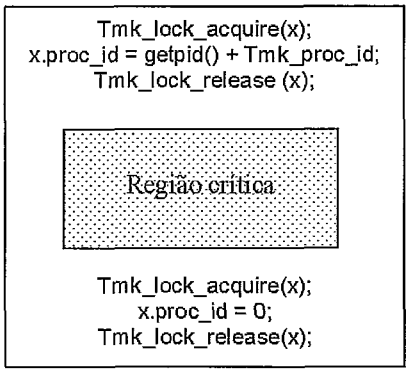


Figura 4.9: Processo de sincronização no Aurora DSM

Principais Problemas Encontrados Durante a Sincronização

Como os problemas encontrados na sincronização constituíram o maior obstáculo para a adaptação do sistema ao software DSM TreadMarks, optou-se por

apresentá-los, aqui, de forma destacada, diferentemente dos apresentados em seções anteriores.

Para efetuar a sincronização do sistema, é preciso determinar:

- As regiões críticas do programa;
- As variáveis compartilhadas que precisam de proteção, baseado nos critérios da seção anterior; e
- As variáveis compartilhadas que são lidas e escritas juntas de forma que possam usar o mesmo número de *lock*.

Como Aurora é um sistema de grande porte e em arquiteturas de memória centralizada acessos à memória têm um custo baixo, alguns problemas foram encontrados para determinar os items citados anteriormente:

1. O programa não tem regiões críticas bem definidas, que possam ser delimitadas por *lock/unlock* na versão DSM;
2. As variáveis compartilhadas estão espalhadas por todo o programa; e
3. As variáveis protegidas, na versão original, que utilizam o mesmo número de *lock*, são usadas em pontos diferentes do programa, provocando a necessidade de se fazer inúmeras operações de *lock/unlock*.

O sistema possui duas estruturas compartilhadas principais, `struct worker` e `struct node`, que representam, respectivamente, os *workers* e os nós de execução. São estruturas grandes, que definem algumas variáveis de *lock* para proteger as variáveis compartilhadas de múltiplos escritores. Por exemplo, a Figura 4.10 apresenta um trecho de `struct_worker`, onde a variável de *lock* `work_wanted_lock`, está associada a alguns dados da estrutura (indicados por *protected*), utilizados quando o *worker* está procurando tarefa para executar. Contudo, estas variáveis protegidas são também utilizadas, separadamente, em outros pontos do programa e com outras finalidades, provocando a necessidade de se incluir inúmeros pares de *lock/unlock*. Estes, por sua vez, atrasam a execução e provocam tráfego de mensagens na rede, como já foi visto anteriormente. De modo similar, isto acontece com `struct node`.

```
struct worker {
    volatile BOOL work_given;
    LOCKTYPE(work_given_lock);          //Tmk_linux
    BOOL cut;                           //protected
    BOOL i_am_working;                  //protected
    BOOL branch_is_cut;                 //protected
    LOCKTYPE(wants_place_lock);         //Tmk_linux
    BOOL wants_place;
    WID work_wanted_by;
    LOCKTYPE(work_wanted_by_lock);      //Tmk_linux
    struct node *left_root;             /* points to my_left_root */
    struct node *interest_node;         /* when trying voluntary suspension */
    LOCKTYPE(work_node_lock);           //Tmk_linux
    struct node *work_node;
    struct node *subtree_root;          //protected
    struct node *cut_node;              //protected
    struct node *current_node;          /* debug */
    struct node *sentry_node;           //protected
    struct alternative *alt;            //protected
    int subtree_level;                  //protected
    LOCKTYPE(work_sync_lock);
    LOCKTYPE(work_wanted_lock);
    LOCKTYPE(ref_lock);
    BOOL has_nonspec_work;              //protected
    short interrupt_flags;
    LOCKTYPE(interrupt_flags_lock);     //Tmk_linux
    LOCKTYPE(richness_lock);            //Tmk_linux
    int richness;
    struct node *public_nonspec_work;   /* link of mandatory work */
}
```

Figura 4.10: `Struct worker`

4.1.4 Liberação de Memória-compartilhada e Término do Sistema

O TreadMarks provê as rotinas `Tmk_free(ptr)` e `Tmk_exit(status)` que, respectivamente, liberam a memória-compartilhada alocada através da chamada à `Tmk_malloc(size)` e terminam o programa. Assim, de forma similar à implementação original, no final da computação da versão DSM, o processo mestre libera as áreas não mais necessárias (`Tmk_free(ptr)`) e cada processo encerra sua execução (`Tmk_exit(status)`).

4.2 Características do Sistema Aurora e Problemas Adicionais Encontrados

4.2.1 Complexidade do Software

Quantidade de linhas, funções e arquivos

A versão inicial do sistema Aurora empregada na tese, adaptada para a execução em diversas plataformas, possui 130.462 linhas e 477 arquivos. Destes, 42.387 linhas e 125 arquivos compõem a porção efetivamente utilizada neste trabalho, que faz uso de 225 funções.

Utilização de macros

O sistema faz uso de um número significativo de macros, principalmente para implementar a interface entre o escalonador e a máquina prolog, assunto abordado no Capítulo 3. E, em sua maioria, estas macros são implementadas com quantidades excessivas de código, o que além de dificultar o entendimento da aplicação, prejudica a operação de depuração, que tem por característica não mostrar o conteúdo das mesmas.

4.2.2 Mecanismos de Espera Ocupada (Busy Waiting)

Em diversas partes do código são empregados mecanismos de espera ocupada (*busy waiting*), através dos quais um programa permanece em *loop* até que uma determinada condição se torne verdadeira. Em memória-compartilhada, esses mecanismos não causam muitos transtornos, uma vez que os acessos à variável condicional do *loop* são feitos à memória local comum a todos os processadores. Mas, em sistemas que utilizam TreadMarks, e conseqüentemente operam sobre arquiteturas de memória-distribuída, acessos consecutivos à variáveis condicionais compartilhadas correspondem a trocas de mensagens pela rede para a obtenção de seus valores atuais. Observe alguns dos exemplos encontrados no sistema Aurora:

```
(a) while (!glob->ready); /* possible infinite loop */
```

No sistema, o processo mestre é responsável pela inicialização de todo o ambiente. Enquanto as configurações iniciais são feitas, os processos escravos devem permanecer em estado de espera, até receberem um sinal de que podem começar sua computação. Essa condição é assegurada pela variável compartilhada

`glob->ready`, lida repetidamente por todos os processos escravos, e tornada verdadeira pelo processo mestre, quando o mesmo termina de criar o ambiente. Nesse momento, todos os outros processos são informados de que já podem começar a procurar trabalho para executar, pois a condição que esperavam ocorreu, ou seja, `glob->ready` se tornou verdadeira. Esta caracteriza uma variável de um escritor e diversos leitores.

Em TreadMarks, uma implementação com comportamento similar à espera ocupada seria:

```
#ifdef Tmk_linux

LOCK(glob->ready_lock);
while(!glob->ready_lock){
    UNLOCK(glob->ready_lock);
    TMK_SLEEP;    /* TCF: macro similar à função sleep do unix */
    LOCK(glob->ready_lock);
}

UNLOCK(glob->ready_lock);

#endif Tmk_linux
```

O problema dessa implementação é a quantidade de *locks* e *unlocks* consecutivos que são necessários para se garantir a leitura do valor correto da variável. Em TreadMarks, cada *lock* é associado a um processo, denominado gerente (*manager*), que controla os últimos acessos feitos aos *locks* sob sua responsabilidade. Dependendo da localização do gerente e do último processo que o obteve, uma operação de *lock* no sistema, pode corresponder aos seguintes passos: (1) envio de uma mensagem ao seu gerente solicitando identificação do último processo que obteve o *lock* solicitado, (2) encaminhamento da mensagem ao processo identificado, (3) mensagem deste ao requisitante informando os números das páginas atualizadas, (4) mensagem com solicitação dos *diffs* dessas páginas e (5) mensagem contendo os *diffs*. Como se pode observar, essa operação executada dentro de um *loop*, como ocorre no exemplo acima, pode inviabilizar uma utilização eficiente do sistema.

A solução em TreadMarks, para casos como esse, é relativamente simples. Observe que a espera ocupada na variável `glob->ready` atua, dentro do sistema, como uma barreira. Assim, basta substituir o código mencionado anteriormente pelo mecanismo de barreira provido pelo TreadMarks: `void Tmk_barrier(unsigned id)`.

`(b) while(!my_worker->work_given)`

Cada *worker* no sistema é representado por uma estrutura de dados denominada `struct worker`, que é utilizada como meio de comunicação entre eles. Assim, no processo de procurar tarefa, deve-se percorrer alguns campos dessa estrutura de cada um dos *workers*, verificando: (a) se em algum deles há tarefas para executar, (b) se existe trabalho que não seja especulativo e, em caso afirmativo, (c) qual deles é o *worker* que possui a maior quantidade de tarefas para ceder. Feita a escolha, o processo deve preencher sua identificação no campo designado para esta finalidade da estrutura do *worker* que possui a tarefa desejada. A partir de então, deve permanecer em estado de espera até que o outro *worker* lhe dê permissão para executar e lhe envie o trabalho solicitado. Essa permissão é dada através da variável `work_given` da estrutura `worker`, que indica que a tarefa já foi concedida e o trabalho já pode ser iniciado.

Mais uma vez, como a variável `work_given` deve ser compartilhada para que todos os processos possam enxergá-la, na versão do sistema Aurora com TreadMarks, essa situação pode gerar tráfego indesejado na rede. Esta variável é caracterizada como de um escritor e um leitor.

Para resolver esse problema na versão TreadMarks, uma solução seria implementar uma fila de processos nos nós, como mostrada na Figura 4.11. Assim, o *worker* interessado em adquirir trabalho registraria interesse na fila do nó com tarefas disponíveis, pedindo que uma mensagem lhe fosse enviada quando o trabalho já pudesse ser executado.

`(c) while (((node)->interested_workers_count))`

A variável `interested_workers_count` é um campo da estrutura de dados `struct node`, apresentada no capítulo 3, que registra a quantidade de *workers* interessados no nó em questão. Quando um determinado *worker* está fazendo *backtracking* e encontra um nó da árvore de execução cujo valor de

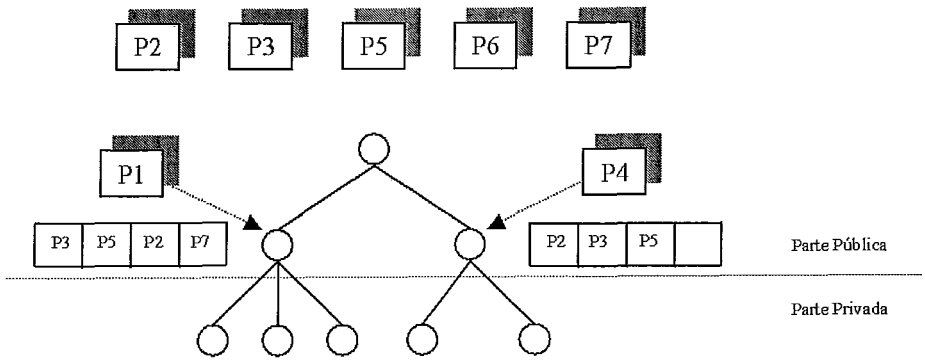


Figura 4.11: Implementação de fila de processos nos nós para obter tarefa

`interested_workers_count` é maior que 1, não pode remover este nó. Portanto, fica neste *loop* aguardando que outros *workers* decrementem esta variável, seja adquirindo trabalho deste nó ou desistindo. Novamente, esta variável deve ser compartilhada por todos os processos, e a leitura repetida na versão de Aurora com TreadMarks pode ser prejudicada. Esta é uma variável de um leitor e diversos escritores.

Capítulo 5

Resultados

Para avaliar o comportamento do sistema desenvolvido durante a pesquisa, Aurora DSM, foram realizados diversos testes. Nesse capítulo, são apresentados os resultados obtidos nas execuções, e as devidas comparações com a versão original, quando as mesmas fizerem-se necessárias. Os testes foram realizados no *cluster* do Laboratório de Inteligência Artificial (LabIA), do Centro de Tecnologia da COPPE/UFRJ. O mesmo é composto por oito computadores com processador Intel Pentium 4, memória principal de 1GByte e sistema operacional Linux (RedHat 7.3).

5.1 Aplicação de Teste

A aplicação de teste escolhida foi o programa queensau.pl, cujo objetivo é encontrar soluções para o problema clássico de Inteligência Artificial (IA) das N-rainhas, que consiste em dispô-las num tabuleiro de xadrez N x N, sem que as mesmas consigam se atacar. Tal condição é alcançada se não houver duas ou mais rainhas numa mesma linha, coluna ou diagonal. Nos testes realizados no sistema Aurora, o programa busca todas as soluções possíveis para um tabuleiro 9x9, que produz 352 soluções e, conseqüentemente, uma grande quantidade de paralelismo OU.

5.2 Análise de Resultados

As medições foram feitas executando-se dez vezes a versão original com 1, 2, 4 e 8 processos em 1 máquina e a versão DSM com 1 e 2 processos em 1 e 2 máquinas. Foram analisados três medidas de tempo, definidas como se segue:

- **Real**: tempo total decorrido entre o início do processo e o seu término;

- **Usuário**: tempo de processador ocupado pelo processo;
- **Sistema**: tempo decorrido em operações do sistema operacional.

A Tabela 5.1 apresenta os tempos reais de execução encontrados nas duas versões quando executadas com 1 e 2 processos em apenas 1 máquina.

Execução	Tmk_1P_1M	Tmk_2P_1M	Ori_1P_1M	Ori_2P_1M
1	00:55,791	06:55,585	00:01,874	00:10,291
2	00:55,659	08:58,324	00:01,863	00:09,150
3	00:55,659	07:46,473	00:01,923	00:12,000
4	00:55,747	07:00,624	00:01,866	00:10,473
5	00:55,661	09:00,253	00:01,865	00:10,200
6	00:55,715	08:05,164	00:01,865	00:09,600
7	00:55,661	07:56,484	00:01,864	00:10,650
8	00:55,656	07:13,353	00:01,869	00:11,850
9	00:55,652	08:57,673	00:01,864	00:10,006
10	00:55,709	08:26,503	00:01,867	00:09,450
Tempo Médio	00:55,691	08:02,044	00:01,872	00:10,367

Tabela 5.1: Tempo Real em 1 Máquina

Comparando-se as médias de tempo de execução de 1 processo em 1 máquina, das versões original e DSM, vê-se que a versão DSM teve uma perda expressiva de desempenho em relação à primeira, sendo aproximadamente 29,3 vezes mais lento do que a versão original.

De forma semelhante, comparando-se as médias de tempo de execução de 2 processos em 1 máquina, das versões original e DSM, vê-se que a perda de desempenho desta em relação a primeira é ainda maior, cerca de 46,2 vezes mais lenta.

Nas Tabelas 5.2 e 5.3, que mostram, respectivamente, os tempos de usuário e de sistema, observa-se o mesmo efeito.

Em números absolutos, novamente, a versão DSM possui tempos mais elevados, embora a relação Tempo DSM/Tempo Original tenha diminuído. Assim, a versão DSM é cerca de 15,4 vezes mais lenta que a versão original para 1 processo/1 máquina, e cerca de 2,7 vezes mais lenta para 2 processos/1 máquina.

A maior discrepância, nos resultados, é encontrada nos números referentes aos tempos consumidos pelos processos na execução de tarefas do sistema operacional. Observando-se os dados da Tabela 5.3, vê-se que, na versão original, os tempos de sistema são praticamente desprezíveis, o que explica a diferença

Execução	Tmk_1P_1M	Tmk_2P_1M	Ori_1P_1M	Ori_2P_1M
1	00:27,520	00:27,450	00:01,840	00:10,260
2	00:28,090	00:27,270	00:01,850	00:09,140
3	00:27,610	00:28,460	00:01,850	00:11,980
4	00:27,990	00:27,600	00:01,860	00:10,450
5	00:27,260	00:28,100	00:01,840	00:10,200
6	00:28,030	00:27,330	00:01,860	00:09,600
7	00:27,940	00:28,050	00:01,850	00:10,650
8	00:28,320	00:28,220	00:01,850	00:11,840
9	00:27,450	00:27,980	00:01,840	00:10,010
10	00:27,950	00:27,780	00:01,850	00:09,440
Tempo Médio	00:27,816	00:27,824	00:01,849	00:10,357

Tabela 5.2: Tempo de Usuário em 1 Máquina

comparação com a versão DSM, que é 2325 vezes mais lenta para 1 processo/1 máquina e 3133 vezes mais lenta para 2 processos/1 máquina.

Execução	Tmk_1P_1M	Tmk_2P_1M	Ori_1P_1M	Ori_2P_1M
1	00:28,160	00:28,600	00:00,020	00:00,010
2	00:27,570	00:28,750	00:00,010	00:00,010
3	00:28,050	00:28,140	00:00,020	00:00,020
4	00:27,750	00:28,440	00:00,000	00:00,030
5	00:28,390	00:27,930	00:00,010	00:00,000
6	00:27,690	00:28,610	00:00,000	00:00,000
7	00:27,720	00:27,880	00:00,010	00:00,000
8	00:27,340	00:27,810	00:00,010	00:00,010
9	00:28,180	00:27,960	00:00,020	00:00,000
10	00:27,750	00:28,260	00:00,020	00:00,010
Tempo Médio	00:27,860	00:28,238	00:00,012	00:00,009

Tabela 5.3: Tempo de Sistema em 1 Máquina

As tabelas 5.4 e 5.5 mostram a percentagem de tempo de usuário e de sistema relativos ao tempo total gasto (tempo real), em cada uma das versões.

Tempo	1P	2P	4P	8P
Real	100,00%	100,00%	100,00%	100,00%
Usuário	98,77%	99,90%	99,97%	99,98%
Sistema	0,64%	0,09%	0,04%	0,01%

Tabela 5.4: Percentagem (Aurora Original)

Ratificando, na versão original, é extremamente mínimo o tempo gasto em chamadas de sistema. E os tempos de usuário e real praticamente se sobrepõem.

Tempo	1P	2P_1M	2P_2M
Real	100%	100%	100%
Usuário	50%	6%	4%
Sistema	50%	6%	4%

Tabela 5.5: Percentagem (Aurora DSM)

Já os resultados obtidos, para a versão DSM, podem ser explicados pela divergência na forma como o programa está estruturado e o modo de operação do TreadMarks. Como se pôde ver no Capítulo 4, adaptar Aurora para um ambiente de software DSM não foi uma tarefa trivial. Por exemplo, o mecanismo de espera ocupada, empregado em grande quantidade ao longo do programa, tem responsabilidade nas três medidas de tempo analisadas. Para relembrar, observe na Figura 5.1 uma amostra encontrada no código do programa, relativa ao mecanismo citado:

```
#ifdef Tmk_linux\\
LOCK(my_worker->interest_lock);\\
\\
while(!(my_worker->interested_workers_count)){\\
  UNLOCK(my_worker->interest_lock);\\
  TMK_SLEEP;     /* TCF: macro similar à função sleep do unix */\\
  LOCK(my_worker->interest_lock);\\
}\\
\\
UNLOCK(my_worker->interest_lock);\\
#endif Tmk_linux\\
```

Figura 5.1: Mecanismo de espera ocupada

Os mecanismos de espera ocupada (*busy waiting*) caracterizam-se por consumir muito tempo de processador, sem estar realizando tarefas de interesse da aplicação. O uso em larga escala desse mecanismo, no sistema Aurora, explica os números elevados obtidos de tempo de usuário, na comparação com a versão original. Além disso, como a estrutura `my_worker` é compartilhada entre os processos, para efetuar uma leitura em um de seus campos, é preciso buscar o seu valor atualizado, o que é feito pela operação de *lock* na variável. Enquanto seu conteúdo é diferente de zero, o processo deve permanecer em *loop* (*busy waiting*), realizando operações seguidas de *lock* e *unlock* na variável, que na prática correspondem a mensagens na rede para encontrar o valor atualizado. Esse tempo é computado nas medidas obtidas do

tempo real do processo. Por último, para evitar que o processo congestione a rede ou ocupe o processador, demasiadamente, por estar em *loop*, introduziu-se chamadas de sistema `sleep()`. Estas são as maiores responsáveis pelos altos tempos de sistema do processo, muito embora o próprio TreadMarks também faça chamadas de sistema que contribuem para este número elevado.

Assim, todos os tempos computados são afetados pelo mecanismo de espera ocupada utilizado por todo o sistema. Entretanto, considerando-se a percentagem de cada tempo na execução, percebe-se que os tempos de sistema e usuário não são tão grandes. Voltando-se à Tabela 5.5, na execução com apenas 1 processo, os tempos de usuário e sistema somam 100% do total, porque não há troca de mensagens (que seria computada no tempo real). Já nas execuções com 2 processos, os mesmos tempos somados não passam de 12%. Desse modo, aproximadamente 88% do tempo restante é gasto em comunicação entre os processos, minimizando as interferências das operações de sistema e de espera ocupada, citadas nesta seção.

Uma visão mais acurada dos tempos de comunicação pode ser obtida através da análise da quantidade de operações de *lock* realizada pelo sistema. As Tabelas 5.6 e 5.7 mostram os números extraídos da execução para as versões original e DSM.

	1P_1M	2P_1M	4P_1M	8P_1M
Locks	2.832	4.848	5.990	8.347
Proporção	1,0	1,7	2,1	2,9

Tabela 5.6: Operações de *Lock* (Aurora Original)

	1P_1M	2P_1M	2P_2M
Locks	4.963.695	34.138.984	24.305.132
Proporção	1	7	5

Tabela 5.7: Operações de *Lock* (Aurora DSM)

Nos dois, a quantidade de *locks* aumenta com a inclusão de mais processos, mas enquanto na versão original esta relação é de 1,7 quando se passa de 1 para 2 processos, na versão DSM esta relação tem um salto de 7. Além dos problemas já mencionados, o número de *locks* também é influenciado pela diferença no código dos processos mestre e escravo. No sistema, na presença de mais de um processo, o código executado pelo mestre é diferente do executado pelo escravo. Este último é caracterizado pela presença de muitas primitivas de sincronização.

A proporção de *locks* entre as duas versões (Original e DSM) pode ser vista na Tabela 5.8.

	1P_1M	2P_1M
Original	2.832	4.848
DSM	4.963.695	34.138.984
Proporção	1.753	7.042

Tabela 5.8: Relação de *Locks* (Original X DSM)

5.3 Análise de Escalabilidade do Sistema

Dando prosseguimento à análise dos resultados, a Tabela 5.9 apresenta o desempenho do sistema Aurora original, para 1, 2, 4 e 8 processos em 1 máquina.

Execução	Ori_1P_1M	Ori_2P_1M	Ori_4P_1M	Ori_8P_1M
1	00:01,874	00:10,291	00:28,716	02:05,632
2	00:01,863	00:09,150	00:34,306	01:48,598
3	00:01,923	00:12,000	00:26,400	02:28,208
4	00:01,866	00:10,473	00:35,205	02:06,467
5	00:01,865	00:10,200	00:31,960	02:17,717
6	00:01,865	00:09,600	00:39,406	01:31,626
7	00:01,864	00:10,650	00:22,799	02:01,316
8	00:01,869	00:11,850	00:30,416	00:56,681
9	00:01,864	00:10,006	00:42,857	02:51,418
10	00:01,867	00:09,450	00:32,667	02:14,077
Tempo Médio	00:01,872	00:10,367	00:32,473	02:02,174

Tabela 5.9: Tempos Reais (Aurora Original)

Veja que, conforme se introduz mais processos ao sistema, maior é o tempo de execução.

Observe, agora, a mesma análise na versão DSM. A Tabela 5.10 apresenta tempos crescentes de execução real na medida em que se adiciona processos ao sistema.

As Tabelas 5.11 e 5.12 mostram as perdas de desempenho correspondentes às Figuras 5.2 e 5.3 de tempo real, para cada uma das versões.

A perda maior na versão DSM está, novamente, no tempo real. Entretanto, os tempos de usuário e sistema, praticamente, não são alterados. Já na versão original, a perda existente em todos os tempos é exponencial, e constitui uma fragilidade do sistema.

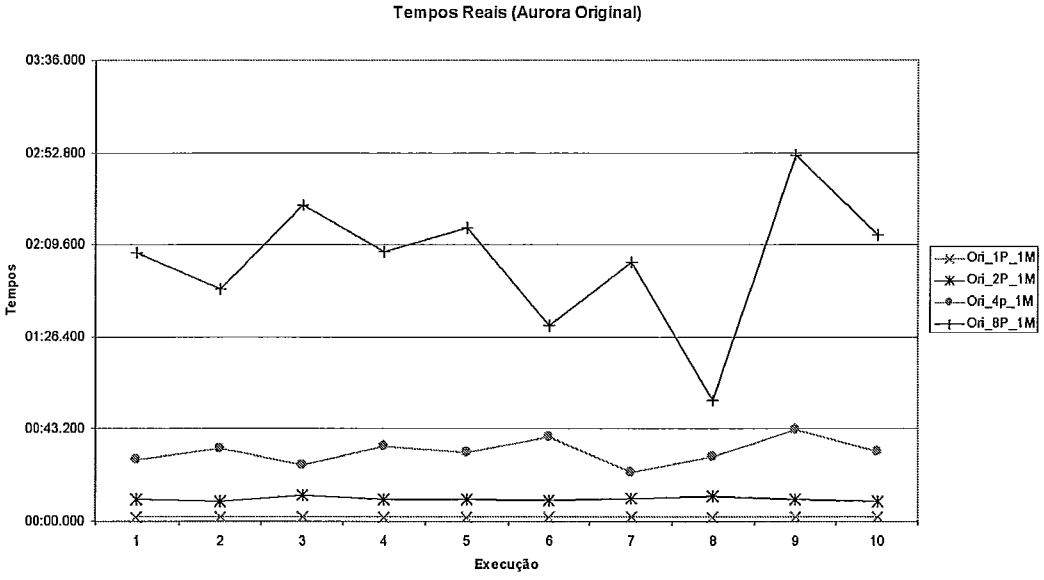


Figura 5.2: Tempos reais (Aurora Original)

Execução	Tmk_1P_1M	Tmk_2P_1M	Tmk_2P_2M
1	00:55,791	06:55,585	11:48,321
2	00:55,659	08:58,324	11:27,126
3	00:55,659	07:46,473	10:25,427
4	00:55,747	07:00,624	13:52,206
5	00:55,661	09:00,253	11:50,056
6	00:55,715	08:05,164	12:27,013
7	00:55,661	07:56,484	12:16,333
8	00:55,656	07:13,353	08:42,746
9	00:55,652	08:57,673	10:28,572
10	00:55,709	08:26,503	13:18,636
Tempo Médio	00:55,691	08:02,044	11:39,644

Tabela 5.10: Tempos Reais (Aurora DSM)

	1P	2P	4P	8P
Real	1,00	5,54	17,35	65,26
Usuário	1,00	5,60	17,56	66,06
Sistema	1,00	0,75	1,00	0,75

Tabela 5.11: Perda de Desempenho (Aurora Original)

As Figuras 5.4 e 5.5, com as linhas de tendência para cada uma das versões, mostram a limitação de escalabilidade existente na versão original do sistema Aurora, e que se reflete parcialmente na versão DSM.

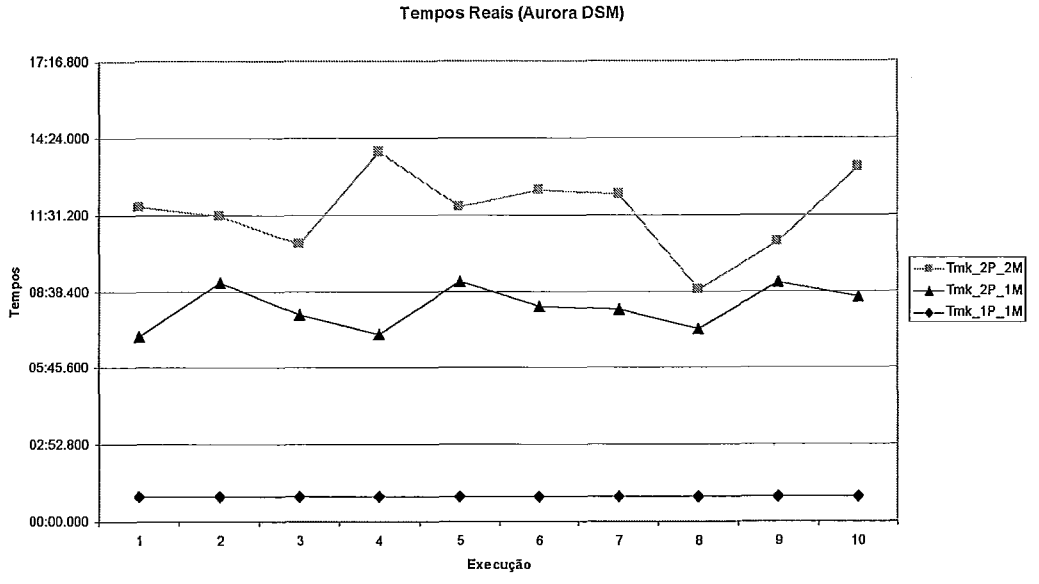


Figura 5.3: Tempos reais (Aurora DSM)

	1P_1M	2P_1M	2P_2M
Real	1,00	8,66	12,56
Usuário	1,00	1,04	1,00
Sistema	1,00	1,06	1,01

Tabela 5.12: Perda de Desempenho (Aurora DSM)

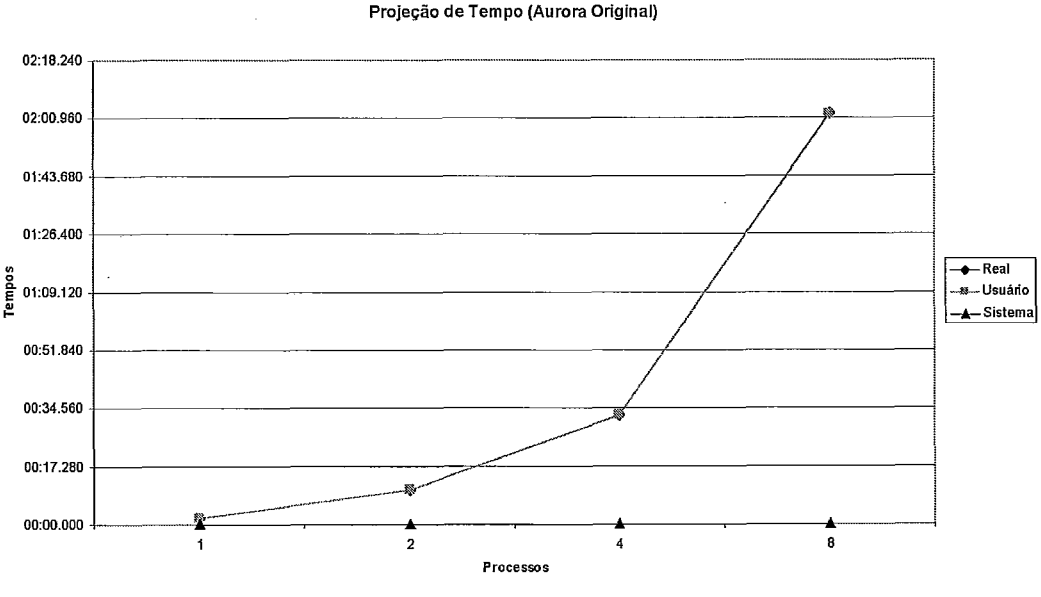


Figura 5.4: Projeção de tempo (Aurora Original)

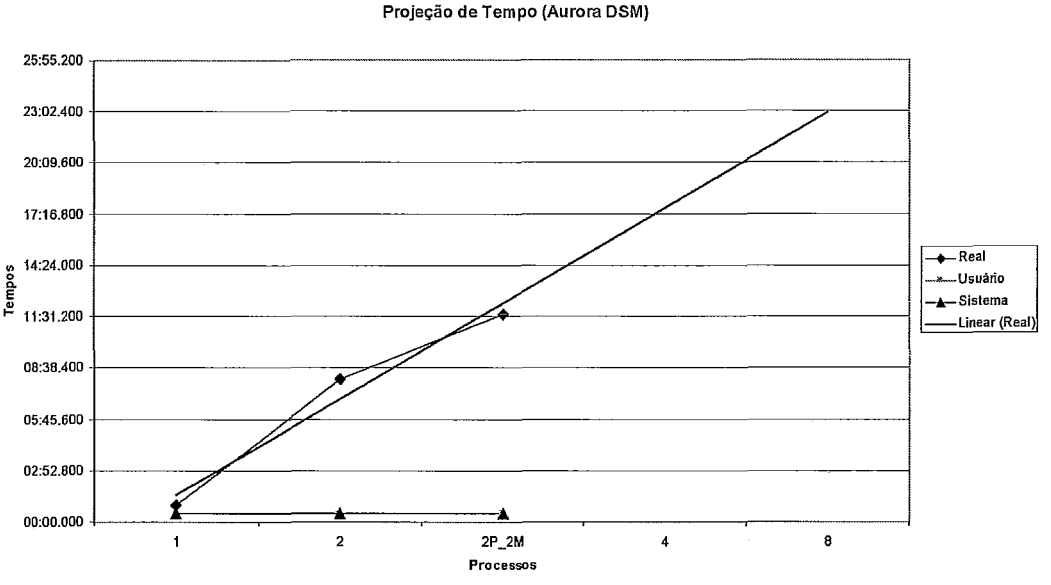


Figura 5.5: Projeção de tempo (Aurora DSM)

Capítulo 6

Conclusões

Os softwares DSM despontaram como uma ferramenta importante na computação paralela, pois possibilitaram a extração das potencialidades existentes nas arquiteturas de memória-compartilhada e distribuída. Assim, mantiveram a facilidade de programação encontrada nos sistemas de memória-compartilhada, escondendo o mecanismo de troca de mensagens exigido no nível de hardware. Além disso, com a memória-distribuída em nível físico, conservaram a vantagem de escalabilidade de tais arquiteturas. Além do mais, permitiram a continuidade dos sistemas projetados para ambientes de memória-compartilhada, antes gradualmente restritos pela arquitetura.

Ao longo dos anos, diversos sistemas DSM foram desenvolvidos e muitas pesquisas objetivaram explorar suas vantagens e características. Os resultados encontrados, até então, eram promissores. Muitas aplicações foram e são beneficiadas com o seu uso, muitos pesquisadores tiveram e têm suas tarefas facilitadas pelos motivos já mencionados. Não há questionamento quanto a sua importância para a computação.

Entretanto, no trabalho de pesquisa dessa tese, observou-se que poucos eram os resultados encontrados para sistemas de grande porte que utilizam uma grande quantidade de sincronização. E, nas referências descobertas, não havia um estudo sobre o processo de se adaptar um sistema dessa escala para um software DSM, como o utilizado nessa tese. Ademais, não existiam quaisquer medidas de desempenho que mostrassem o comportamento da aplicação.

Assim, esta pesquisa foi desenvolvida com o intuito de preencher uma lacuna importante, ainda existente nesse assunto. Pelas dificuldades experimentadas em pesquisas anteriores, de certo modo, esperava-se que muitos obstáculos fossem

encontrados ao longo do caminho. E, os resultados comprovaram o que antes era apenas teoria. Migrar um sistema para um ambiente DSM é possível, mas nem sempre é recomendável. Como pôde ser visto nessa pesquisa, as características da aplicação são fatores importantes nessa decisão e devem ser avaliados, cuidadosamente. A forma como um sistema está estruturado pode ir de encontro ao modo como o software DSM trabalha. Nesse caso, a solução é reestruturá-lo, de forma a minimizar os conflitos identificados.

O sistema Aurora apresentou diversos problemas no processo de sua migração para o software DSM TreadMarks. O fato de não haver regiões concentradoras de operações em variáveis compartilhadas, induziu ao uso excessivo de primitivas de sincronização e, conseqüentemente, de troca de mensagens entre os processos. A utilização de sincronização apenas nas variáveis de múltiplos escritores exigiu que se fizesse um levantamento de todas as variáveis compartilhadas do sistema, de forma a sincronizá-las em todas as operações de escrita e leitura. O uso demasiado de mecanismos de espera ocupada também comprometeu o sucesso da migração. Além disso, o próprio sistema Aurora apresentou problemas de escalabilidade que podem comprometer ainda mais a execução. Para resolver esses problemas, é preciso que se projete o sistema de modo que o mesmo possa atender as necessidades do software DSM. A implementação das soluções propostas, para cada um dos problemas identificados nessa tese, pode contribuir para a obtenção de melhores desempenhos das execuções.

Este trabalho foi desenvolvido ao longo de dois anos ininterruptos, e foram dispensados muitos esforços no sentido de conseguir pôr o sistema para funcionar com o software DSM TreadMarks. O porte do sistema Aurora, a sua estrutura em conflito com o software DSM, a legibilidade precária do código e a ausência de uma documentação do mesmo foram apenas alguns dos obstáculos a serem superados ao longo do caminho.

Conclui-se, portanto, que o baixo desempenho extraído das medições feitas até agora, não compensa o esforço necessário para se migrar aplicações do porte e estilo do sistema Aurora para ambientes DSM, a menos que se mude a forma como o software DSM opera ou se invista tempo em reescrever e re-projetar o software que foi escrito para memória centralizada. A combinação de Aurora e TreadMarks, por exemplo, requer mudanças estruturais muito complexas, além das etapas normalmente executadas para outros sistemas menores.

6.1 Propostas Futuras

No sentido de melhorar os resultados apresentados, discutimos algumas soluções relacionadas ao sistema Aurora e ao software DSM TreadMarks. Um primeiro passo, é avaliar o problema de escalabilidade encontrado na versão original que, conseqüentemente, se reflete na versão DSM. Em seguida, pode-se implementar as soluções propostas para os problemas identificados ao longo da pesquisa dessa tese. Há a necessidade de se fazer um estudo aprofundado de todas as estruturas de dados e códigos do sistema, de forma a reestruturá-lo para suportar o software DSM desejado. Por exemplo, definir regiões críticas no sistema, e tentar concentrar nessas áreas de código todos os acessos, sejam de leitura ou escrita, a variáveis compartilhadas. Avaliar as estruturas compartilhadas que mais degradem a execução e, nesses casos, efetuar troca explícita de mensagem. Essa solução visa diminuir a quantidade de mensagens desnecessárias na rede, como ocorre na implementação de espera ocupada. Assim, teria-se um modelo híbrido, onde estariam presentes os dois conceitos de memória: compartilhada e distribuída, com algumas variáveis na área de memória privada e outras na área de memória-compartilhada.

Sob o ponto de vista do software DSM, é necessário que este torne a tarefa do programador mais transparente, de forma que o programa escrito para memória centralizada sofra modificações mínimas. Objetiva-se com isso que, a migração para ambiente distribuído seja mais amena para o usuário e ao mesmo tempo mantenha um grau razoável de eficiência. Diversos trabalhos vêm caminhando no sentido de prover maior eficiência para manter as memórias coerentes tais como utilização de protocolos adaptativos [5], técnicas de prefetching [13, 14], adaptação a padrões de compartilhamento [64, 26], tolerância à latência [65], previsão de aquisição de locks [78], entre outros. Porém pouco ou nenhum trabalho tem se concentrado em tornar a tarefa de migração mais transparente.

Isto poderia ser conseguido se o software DSM fosse provido de mecanismos de *tracing* de execução que: (a) observassem todos os acessos a dados compartilhados dentro e fora de seção crítica, e (b) pudessem descobrir quando e para que processador enviar as modificações locais. Nesse caso, não haveria necessidade do usuário, explicitamente, empregar alguma forma de API diferente da utilizada no modelo de programação baseado em memória centralizada.

Apêndice A

Um Exemplo de Código TreadMarks - quicksort.c

```
#include <stdio.h>
#include "Tmk.h"
extern char *optarg;
typedef enum {false=0, true=1} boolean;

int tam_vetor= 0;
int* vetor = NULL;

struct trab {
   int inicio;
   int fim;
   struct trab * prox;
};

struct fila {
   int rodando;
   struct trab* inicio;
   struct trab* fim;
} *tarefas;

/*
 * inicializar - carrega o vetor a ser ordenado e
 * prepara a lista de tarefas.
 */
void inicializar(int tamanho) {
   int i;

   printf("(%d) Inicializar tamanho=%d\n", getpid(), tamanho);
   srand(time(NULL));

   tam_vetor = tamanho;
   vetor = Tmk_malloc(tam_vetor * sizeof(int));
```

```
   if (vetor == NULL) {
      printf("Nao ha memoria compartilhada suficiente!\n");
      Tmk_exit(1);
   }

   for (i = 0; i < tam_vetor; i++) {
      int x = 0;
      boolean ok;
      do {
         int j;

         x = rand() % (tam_vetor * 100);
         ok = true;
         for (j = 0; j < i && ok; j++) {
           if (vetor[j] == x) {
              ok = false;
           }
         }
      } while(!ok);
      vetor[i] = x;
   }

   Tmk_distribute(&tam_vetor, sizeof(tam_vetor));
   Tmk_distribute(&vetor, sizeof(vetor));

   tarefas = Tmk_malloc(sizeof(struct fila));
   if (tarefas == NULL) {
      printf("Nao ha memoria compartilhada suficiente!\n");
      Tmk_exit(1);
   }

   tarefas->rodando = 0;
   tarefas->inicio = NULL;
   tarefas->fim = NULL;
   adicionar(0, tam_vetor-1);
   Tmk_distribute(&tarefas, sizeof(tarefas));
} // inicializar(int)

void imprimir_tarefa(struct trab* tarefa) {
   printf("(%d) tarefa[%d,%d]\n",
        getpid(), tarefa->inicio, tarefa->fim);
} // imprimir_tarefa(struct trab*)

/*
 * imprimir_fila - Imprime a fila de execucao
 */
void imprimir_fila(char* nome) {
```

```
   struct trab* tarefa;
   boolean primeiro = true;

   tarefa = tarefas->inicio;
   printf("(%d) %s{", getpid(), nome);
   while (tarefa != NULL) {
      if (!primeiro) {
         printf(", ");
      }
      printf("[%d,%d]", tarefa->inicio, tarefa->fim);
      primeiro = false;
      tarefa = tarefa->prox;
   }
   printf("}\n");
} // imprimir_fila(char*)

/*
 * imprimir_vetor - Imprime o vetor
 */
void imprimir_vetor(char* nome, int inicio, int fim) {
   int i = 0;
   boolean primeiro = true;

//   printf("(%d) imprimir_vetor\n", getpid());
   if (inicio < 0 || fim >= tam_vetor || inicio > fim) {
      printf("(%d) indices invalidos. inicio=%d, fim=%d\n",
            getpid(), inicio, fim);
      Tmk_exit(1);
   }

   printf("(%d) %s[%d,%d] {", getpid(), nome, inicio, fim);
   for (i = inicio; i <= fim; i++) {
      if (!primeiro) {
         printf(", ");
      }
      printf("%d", vetor[i]);
      primeiro = false;
   }
   printf("}\n");
} // imprimir_vetor(int, int)

boolean vetor_ordenado() {
   int i;
   boolean ordenado = true;

   for (i = 1; i < tam_vetor; i++) {
      if (vetor[i] <= vetor[i-1]) {
```

```
            printf("(%d) %d(%d) e %d(%d) estao fora de ordem!!!\n",
                   getpid(), i-1, vetor[i-1], i, vetor[i]);
            ordenado = false;
         }
      }

      return ordenado;
}

/*
 * copia_local - copia a tarefa da memoria
 * compartilhada para a memoria local
 */
struct trab* copia_local(struct trab* tarefa) {
      struct trab* copia;

      copia = (struct tab*)malloc(sizeof(struct trab));
      if (copia == NULL) {
         printf("Nao ha memoria suficiente!\n");
         Tmk_exit(1);
      }

      copia->inicio = tarefa->inicio;
      copia->fim = tarefa->fim;
      copia->prox = NULL;

      return copia;
} // copia_local(struct trab*)

/*
 * extrai_fila - Extrai um elemento da fila de execucao.
 */
struct trab* extrai_fila() {
      struct trab* tarefa;

//    printf("(%d) extrai_fila\n", getpid());
      Tmk_lock_acquire(0);
      if (tarefas->inicio != NULL) {
         struct trab* inicio;
         //imprimir_fila("extraindo tarefa");
         inicio = tarefas->inicio;
         tarefa = copia_local(inicio);
         tarefas->inicio = inicio->prox;

         if (tarefas->inicio == NULL) {
            tarefas->fim = NULL;
         }
```

```
      imprimir_tarefa(tarefa);
      //imprimir_fila("tarefa extraida");
      tarefas->rodando++;
      Tmk_lock_release(0);

      Tmk_free(inicio);
   } else {
      Tmk_lock_release(0);
      tarefa = NULL;
   }

   return tarefa;
} // extrai_fila()

/*
 * pega_tarefa - Pega uma tarefa na fila de tarefas.
 * Indica se deve continuar executando.
 * O processo deve continuar rodando em duas situacoes:
 * 1) Ha tarefa pendente
 * 2) Ha tarefa executando
 *
 * Eh alocada memoria local para a tarefa retirada, que
 * deve ser encerrada utilizando-se terminar()
 */
boolean pega_tarefa(struct trab** tarefa) {
   boolean cont = true;

//   printf("(%d) pega_tarefa\n", getpid());
   if ((*tarefa = extrai_fila()) != NULL) {
      cont = true;
   } else {
      Tmk_lock_acquire(0);
      cont = (tarefas->rodando > 0);
      Tmk_lock_release(0);
   }

   return cont;
} // pega_tarefa(struct trab**)

/*
 * terminar - Encerrar uma tarefa, liberando o espaco em
 * memoria alocado a esta.
 */
void terminar(struct trab* tarefa) {
//   printf("(%d) terminar\n", getpid());
```

```
   Tmk_lock_acquire(0);
   tarefas->rodando--;
   Tmk_lock_release(0);

   free(tarefa);
} // terminar(struct trab*)

/*
 * adicionar - Adiciona uma tarefa a lista de tarefas pendentes.
 */
void adicionar(int inicio, int fim) {
   struct trab* tarefa = NULL;

   printf("(%d) adicionar(%d, %d)\n", getpid(), inicio, fim);

   tarefa = Tmk_malloc(sizeof(struct trab));
   if (tarefa == NULL) {
      printf("Nao ha memoria compartilhada suficiente!\n");
      Tmk_exit(1);
   }

   Tmk_lock_acquire(0);
   tarefa->inicio = inicio;
   tarefa->fim = fim;
   tarefa->prox = NULL;

   if (tarefas->fim != NULL) {
      tarefas->fim->prox = tarefa;
   }

   tarefas->fim = tarefa;

   if (tarefas->inicio == NULL) {
      tarefas->inicio = tarefas->fim;
   }
   Tmk_lock_release(0);
} // adicionar(int, int)

void sort(int inicio, int fim) {
   int i, j, i_x;
   int w, x;

   //printf("(%d) sort(%d, %d)\n", getpid(), inicio, fim);
   //imprimir_vetor("sort inicio", inicio, fim);
   i = inicio;
   j = fim;
```

```
   i_x = (int)((inicio + fim) / 2);
   Tmk_lock_acquire(0);
   x = vetor[i_x];
   Tmk_lock_release(0);
   //printf("(%d) sort: x (%d) = %d\n", getpid(), i_x, x);

   do {
      Tmk_lock_acquire(0);
      for (;vetor[i] < x; i++);
      Tmk_lock_release(0);

      Tmk_lock_acquire(0);
      for (;x < vetor[j]; j--);
      Tmk_lock_release(0);

      if (i <= j) {
         Tmk_lock_acquire(0);
         w = vetor[i];
         vetor[i] = vetor[j];
         vetor[j] = w;
         Tmk_lock_release(0);
        i++;
        j--;
      }
   } while (i <= j);

   //imprimir_vetor("sort fim", inicio, fim);
   if (inicio < j) {
      adicionar(inicio, j);
   }

   if (i < fim) {
      adicionar(i, fim);
   }
} // sort(int, int)

main(int argc, char* argv[]) {
   int c, tamanho = 30;
   while ((c = getopt(argc, argv, "d:")) != -1) {
      switch (c) {
      case 'd':
         tamanho = atoi(optarg);
         if (tamanho < 0) {
           printf("Tamanho tem que ser maior que zero\n");
           exit(1);
         }
         break;
```

```
      }
   }

   Tmk_startup(argc, argv);
   printf("(%d) Tmk_proc_id: %d\n", getpid(), Tmk_proc_id);

   if (Tmk_proc_id == 0) {
      inicializar(tamanho);
      imprimir_vetor("inicial", 0, tam_vetor-1);
   } // if (Tmk_proc_id == 0)

   Tmk_barrier(0);
   while (true) {
      struct trab* tarefa = NULL;

      if (!pega_tarefa(&tarefa)) {
         break;
      }

      if (tarefa != NULL) {
         sort(tarefa->inicio, tarefa->fim);
         terminar(tarefa);
      } // if (tarefa != NULL)
   } // while (true)
   Tmk_barrier(0);

   printf("(%d) Ordenacao encerrada\n");
   if (Tmk_proc_id == 0) {
      imprimir_vetor("final", 0, tam_vetor-1);
      if (vetor_ordenado()) {
         printf("Vetor ordenado corretamente\n");
      }
   } // if (Tmk_proc_id == 0)

   Tmk_exit(0);
} // main(int, char**)
```

Apêndice B

A Interface do TreadMarks

```
/* O número máximo de processos suportados pelo TreadMarks */
#define TMK_NPROCS

/* O número real de processos paralelos em uma execução específica */
extern unsigned Tmk_nprocs;

/* O id do processo, um inteiro no intervalo de 0 ... Tmk_nprocs - 1 */
extern unsigned Tmk_proc_id;

/* O número máximo de páginas de memória-compartilhada*/
#define TMK_NPAGES

/* O tamanho das páginas de memória-compartilhada alocadas pelo TreadMarks */
extern unsigned Tmk_page_size;

/* O número de locks que será fornecido pelo TreadMarks */
#define TMK_NLOCKS

/* O número de barreiras que será fornecido pelo TreadMarks */
#define TMK_NBARRIERS

/* Inicia o TreadMarks e cria os processos remotos */
void Tmk_startup(int argc, char **argv)

/* Termina o processo que realizou essa chamada. Os outros não são afetados */
void Tmk_exit(int status)

/* Bloqueia o processo até que todos os outros cheguem na barreira especificada */
void Tmk_barrier(unsigned id)

/* Bloqueia o processo até que o mesmo adquira o lock especificado */
void Tmk_lock_acquire(unsigned id)

/* Libera o lock especificado */
void Tmk_lock_release(unsigned id)

/* Aloca o número especificado de bytes de memória principal */
char *Tmk_malloc(unsigned size)

/* Libera a memória-compartilhada alocada pela chamada a Tmk_malloc */
char *Tmk_free(char *ptr)

/*Aloca o número especificado de bytes de memória-compartilhada. Com essa
chamada, o bloco de memória não pode ser liberado pela aplicação */
char *Tmk_sbrk(int incr)

/* Distribui o endereço de um bloco de memória para todos os outros processos */
Tmk_distribute(char *ptr, unsigned size)
```

Figura B.1: Interface provida pelo TreadMarks

Referências Bibliográficas

[1] AHUJA, S., CARRIERO, N. and GELERNTER, D. Linda and Friends. *Computer*, 19(8):26–34, 1986.

[2] ALI, K. A. M. and KARLSSON, R. The Muse Or-parallel Prolog Model and its Performance. In *NACLP90*, pages 757–776. MIT Press, October 1990.

[3] ALI, K. A. M. and KARLSSON, R. Performance of Muse on the BBN Butterfly TC2000. In *LNCS 605, PARLE'92 Parallel Architectures and Languages Europe*, pages 603–616. Springer-Verlag, June 1992.

[4] AMZA, C., et al. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 19(2):18–28, February 1996.

[5] AMZA, C., et al. Adaptive Protocols for Software Distributed Shared Memory. In *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, Spring 1999.

[6] APT, K. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[7] AÏT-KACI, H. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[8] BAL, H. E. and TANENBAUM, A. S. Distributed Programming with Shared Data. In *International Conference on Computer Languages*, pages 82–91. IEEE Computer Society Press, 1988.

[9] BBN ADVANCED COMPUTERS INC. Inside the TC2000 Computer, 1990.

[10] BEAUMONT, A. *Scheduling in OR-Parallel Prolog Systems.* PhD thesis, University of Bristol, Department of Computer Science, 1993.

[11] BERSHAD, B. N., ZEKAUSKAS, M. J. and SAWDON, W. A. The Midway Distributed Shared Memory System. In *Proceedings of the COMPCON 93 Conference*, pages 528–537. IEEE Computer Society Press, 1993.

[12] BIANCHINI, R., et al. A Segunda Geração de Computadores de Alto Desempenho da COPPE/UFRJ. In *VII Simpósio Brasileiro de Arquitetura de Computadores, SBAC-PAD*, July 1996.

[13] BIANCHINI, R., et al. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *7th International Conference on Architectural Support for programming Languages and Operating Systems (ASPLOS 7)*, October 1996.

[14] BIANCHINI, R., PINTO, R. and AMORIM, C. L. Data Prefetching for Software DSMs. In *Proc. of the Int'l Conference on Supercomputing'98*, pages 385 – 392, July 1998.

[15] BISANI, R. and RAVISHANKAR, M. Plus: A Distributed Shared-Memory System. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 115–124, Los Alamitos, Calif, 1990. IEEE Computer Society Press.

[16] BUTLER, R., et al. ANLWAM: A Parallel Implementation of the Warren Abstract Machine. Internal Report, Argonne National Laboratory, 1986.

[17] CARLSSON, M. and SZEREDI, P. The Aurora Abstract Machine and its Emulator. SICS Research Report R90005, Swedish Institute of Computer Science, 1990.

[18] CARLSSON, M. et al. SICStus Prolog users manual. Technical report t91:11b, Swedish Institute of Computer Science, 1991.

[19] CARTER, J. B., BENNET, J. K. and ZWAENEPOEL, W. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symp. Operating Systems Principles*, pages 152–164, New York, 1991. ACM Press.

[20] CHAIKEN, D., KUBIATOWICZ, J. and AGARWAL, A. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 314–324. IEEE Computer Society Press, 1994.

[21] CLARK, K. and GREGORY, S. PARLOG: Parallel Programming in Logic. *ACM TOPLAS*, 8(1), January 1986.

[22] CLOCKSIN, W. and MELLISH, C. *Programming in Prolog.* Springer-Verlag, 1981.

[23] CLOCKSIN, W. and MELLISH, C. *Programming in Prolog.* Springer-Verlag, 1986.

[24] COHEN, J. A View of the Origins and Development of Prolog. *Communications of the ACM*, 31(1):26–36, 1988.

[25] COSTA, V. S., WARREN, D. H. D. and YANG, R. Andorra-I: A Parallel Prolog System that Transparently Exploits both AND- and OR-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[26] DE CASTRO, M. C. S. and AMORIM, C. L. Efficient Categorization of Memory Sharing Patterns in Software DSM Systems. In *International Parallel and Distributed Processing Symposium.* ACM&IEEE, 2001. San Francisco, California, USA.

[27] DE KERGOMMEAUX, J. C. and CODOGNET, P. Parallel Logic Programming Systems. *ACM Comput. Surv.*, 26(3):295–336, 1994.

[28] DELP, G., FARBER, D. and MINNICH, R. Memory as a Network Abstraction. *IEEE Network*, pages 34–41, 1991.

[29] DERANSART, P., et al. *Prolog: The Standard: Reference Manual.* Springer Verlag, 1996.

[30] DUTRA, I. C. *Distributing AND- and OR-Work in the Andorra-I Parallel Logic Programming System.* PhD thesis, University of Bristol, Department of Computer Science, February 1995. available at http://www.cos.ufrj.br/~ines.

[31] EWING, L., et al. The Aurora Or-parallel Prolog System. In *FGCS88*, pages 819–830. ICOT, Tokyo, Japan, November 1988.

[32] EWING, L., et al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.

[33] FLEISCH, B. and POPEK, G. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 211–223. ACM Press, 1989.

[34] FRANK, S., et al. The KSR1: Bridging the Gap Between Shared Memory and MPPs. In *Proceedings of the COMPCON 93 Conference*, pages 285–294. IEEE Computer Society Press, 1993.

[35] GEIST, A., et al. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Knoxville, TN, May 1994.

[36] GEYER, C. F. R., VARGAS, P. K. and DUTRA, I. C. Parallelism in Logic Programming. In *Intl. School on Advanced Techniques for Parallel Computation with Applications*, page 35, Natal, RN, Brasil, Sep-Oct 1999.

[37] GUPTA, G. *Multiprocessor Execution of Logic Programs.* PhD thesis, Department of Computer Science, New Mexico State University, Las Cruces, July 1993. http://www.cs.nmsu.edu/lldap/.

[38] GUPTA, G. and JAYARAMAN, B. Analysis of OR-Parallel Execution Models. *ACM Transactions on Programming Languages and Systems*, 15(4):659–680, September 1993.

[39] GUPTA, G. and JAYARAMAN, B. AND-OR Parallelism on Shared Memory Multiprocessors. *Journal of Logic Programming*, 17(1):59–89, 1993.

[40] GUPTA, G., et al. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *ICLP94*, Italy, June 1994.

[41] GUPTA, G., et al. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(4):472–602, 2001.

[42] HAGERSTEN, E., LANDIN, A. and HARIDI, S. DDM - A Cache-Only Memory Architecture. *Computer*, 25(9):44–54, 1992.

[43] HENNESSY, J. L. and PATTERSON, D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 2nd edition, 1996.

[44] HERMENEGILDO, M. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Program in Parallel.* Ph.d. thesis, University of Texas, Austin, 1986.

[45] HERMENEGILDO, M. An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs. In Ehud Shapiro, editor, *ICLP86*, pages 25–39. Springer-Verlag, 1986.

[46] HERMENEGILDO, M. and GREENE, K. The &-Prolog System: Exploiting Independent AND-Parallelism. *New Generations Computing*, 9(3-4):233–257, 1991.

[47] HUANG, Z., et al. Parallel Logic Programming on Distributed Shared Memory System. In *Proceedings of the IEEE International Conference on Intelligent Processing Systems*, October 1997.

[48] IFTODE, L. *Home-Based Shared Virtual Memory.* PhD thesis, Princeton University, Princeton, USA, Jun. 1998.

[49] JAMES, D.V. The Scalable Coherent Interface: Scaling to High-Performance Systems. In *Proceedings of the COMPCON 94 Conference*, pages 64–71, Los Alamitos, Calif, 1994. IEEE Computer Society Press.

[50] KELEHER, P., et al. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Technical report, Rice University, 1990.

[51] KUSKIN, J., et al. The Stanford Flash Multiprocessor. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 302–313. IEEE Computer Society Press, 1994.

[52] LAWRENCE LIVERMORE NATIONAL LABORATORY. Introduction to Parallel Computing. *LLNL Tutorials*, December 2003. Available at http://www.llnl.gov/computing/tutorials/parallel_comp/.

[53] LENOSKI, D., et al. The Stanford DASH Multiprocessor. *Computer*, 25(3):63–79, 1992.

[54] LI, K. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing*, pages 94–101, Los Alamitos, CA, 1988. IEEE Computer Society Press.

[55] LI, K. and HUDAK, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Computer Systems*, 7(4):321–359, Nov. 1989.

[56] LIN, Y. and KUMAR, V. An Execution Model for Exploiting AND-Parallelism in Logic Programs. *New Generation Computing*, 5(4):393–425, 1988.

[57] LIN, Y. and KUMAR, V. And-parallel execution of logic programs on a shared memory multiprocessor: A summary of results. In R. Kowalski and K. Bowen, editors, *In Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141, MA, 1988. MIT Press and Cambridge.

[58] LLOYD, J. W. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[59] LOVETT, T. and THAKKAR, S. The Symmetry Multiprocessor System. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, 1988.

[60] LUCI, S., et al. Reflective-Memory Multiprocessor. In *Proceedings of the 28th IEEE/ACM Hawaii International Conference on System Sciences*, pages 85–94. IEEE Computer Society Press, 1995.

[61] MACHADO, F. B. and MAIA, L. P. *Arquitetura de Sistemas Operacionais.* LTC - Livros Técnicos e Científicos Editora S.A., 2002.

[62] MAPLES, C. and WITTIE, L. Merlin: A Superglue for Multicomputer Systems. In *Proceedings of the COMPCON 90 Conference*, pages 73–81. IEEE Computer Society Press, 1990.

[63] MESSAGE PASSING INTERFACE FORUM. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, University of Tenessee, Knoxville, TN, 1994.

[64] MONNERAT, L. R. and BIANCHINI, R. Efficiently Adapting to Sharing Patterns in Software DSMs. In *HPCA4*, pages 289 – 299, Febrary 1998.

[65] MOWRY, T., CHAN, C. and LO, A. Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory. In *HPDC-4*, pages 300 – 309, Febrary 1998.

[66] O'KEEFE, R. A. *The Craft of Prolog.* MIT Press, 1990.

[67] OVERBEEK, R. A., et al. Prolog on Multiprocessors. Internal Report, Argonne National Laboratory, 1985.

[68] PARALLELTOOLS, L.L.C. Concurrent Programming with TreadMarks. by ParallelTools, L.L.C., 1994.

[69] PONTELLI, E. Adventures in Parallel Logic Programming. http, 1996. Available at http://www.cs.nmsu.edu/~epontell/adventure/.

[70] PONTELLI, E., et al. Improving the Efficiency of Non-Deterministic Independent AND-Parallel Systems. *Computer Languages*, 22(2-3):115–142, 1996.

[71] PONTELLI, E., GUPTA, G. and HERMENEGILDO, M. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the International Parallel Processing Symposium*, pages 564–571, Los Alamitos, CA, 1995. IEEE Computer Society.

[72] PROTIĆ, J., TOMASEVIĆ, M. and MILUTINOVIĆ, V. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):63–79, Summer 1996.

[73] RAMACHANDRAN, U. and KHALIBI, M. Y. A. An Implementation of Distributed Shared Memory. *Software Practice and Experience*, 21(5):443–464, 1991.

[74] RAMKUMAR, B. and KALÉ, L. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In E. Lusk and R. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 313–331, MA, 1989. MIT Press and Cambridge.

[75] RANGARAJAN, M. and IFTODE, L. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 341–352, Atlanta, Georgia, USA, Oct. 2000.

[76] SAMANTA, R. et al. Home-based SVM Protocols for SMP Clusters: Design and Performance. In *Proceedings of the 4th Symposium on High-Performance Computer Architecture*, pages 1–13, Las Vegas, USA, Feb. 1998.

[77] SCHOINAS, I., et al. Fine-Grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference Architectural Support for Programming Languages and Operating Systems*, pages 297–306. ACM Press, 1994.

[78] SEIDEL, C. B., BIANCHINI, R. and AMORIM, C. L. Evaluating the Impact of the Programming Model on the Performance and Complexity of Software DSM Systems. In *International Conference on Parallel Processing*, pages 228–, 1999.

[79] SHAPIRO, E. A Subset of Concurrent Prolog and its Interpreter. Technical report, Weizmann Institute, Rehovot, Israel, 1983.

[80] SHEN, K. *Studies of AND/OR Parallelism in Prolog.* PhD thesis, Computer Laboratory, University of Cambridge, 1992.

[81] SHEN, K. Initial Results from the Parallel Implementation of DASWAM. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming.* The MIT Press, 1996.

[82] STALLINGS, W. *Operating Systems.* Prentice Hall, 2nd edition, 1995.

[83] STERLING, L. and SHAPIRO, E. *The Art of Prolog.* MIT Press, 1986.

[84] STERLING, L. and SHAPIRO, E. *The Art of Prolog.* MIT Press, 1994.

[85] TANENBAUM, A. S. *Organização Estruturada de Computadores.* Prentice/Hall do Brasil, 1992.

[86] TANENBAUM, A. S. *Sistemas Operacionais Modernos.* Prentice-Hall do Brasil, 1995.

[87] UEDA, K. and CHIKAYAMA, T. Design of the Kernel Language for the Parallel Inference Machine. *Computer Journal*, 33(6), 1990.

[88] WARREN, D. H. D. OR-Parallel Execution Models of Prolog. In *TAPSOFT'87, The 1987 International Joint Conference on Theory and Practice of Software Development, Pisa, Italy*, pages 243–259. Springer-Verlag, March 1987.

[89] WESTPHAL, H., et al. The PEPSys Model: Combining Backtracking, AND- and OR-Parallelism. In *The 1987 Symposium on Logic Programming, San Francisco, California*, pages 436–448, Los Alamitos, CA, 1987. IEEE Computer Society.

[90] WILSON, A., LAROWE, R. and TELLER, M. Hardware Assist for Distributed Shared Memory. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 246–255. IEEE Computer Society Press, 1993.

[91] ZHOU, S., STUMM, M. and MELNERNEY, T. Extending Distributed Shared Memory to Heterogeneous Environments. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 30–37. IEEE Computer Society Press, 1990.