

Mestrado em Engenharia Informática

Memória Partilhada Distribuída

João Alexandre Carvalho Pinheiro Leite

Julho 1995

1. Introdução	4
2. Memória Partilhada vs Troca de Mensagens	4
3. Multiprocessadores, Multicomputadores e Memória Partilhada	5
3.1 Multiprocessadores Baseados em Bus	7
3.1.1 Protocolo ‘Snooping Cache’	7
3.2 Multiprocessadores ‘Switched’	7
3.2.1 Protocolo Baseado em Directorias	8
4. Aspectos Arquiteturais	8
4.1 Modelos de Consistência	9
4.1.1 Consistência Atómica	9
4.1.2 Consistência Sequencial	10
4.1.3 Consistência Causal	10
4.1.4 Consistência ‘PRAM’ (Pipelined RAM)	11
4.1.5 Consistência de Cache	11
4.1.6 Consistência de Processador	11
4.1.7 Memória Lenta	12
4.1.8 Consistência Fraca	12
4.1.9 Consistência ‘Release’	13
4.1.10 Consistência de Entrada	14
4.2 Algoritmos Básicos	14
4.2.1 Modelo centralizado	14
4.2.2 Modelo de migração	15
4.2.3 Modelo de replicação de leitura	16
4.2.4 Modelo de replicação total	16
4.3 Dono, Réplicas e sua Localização	17
4.3.1 Gestão Centralizada	17
4.3.2 Gestão Distribuída Fixa	18
4.3.3 Gestão Distribuída Dinâmica	18
4.4 Sincronização	19
4.4.1 Mecanismos Básicos de Sincronização	19
4.4.2 Implementação	20
4.5 Estrutura do Espaço Partilhado	21
5. Sistemas Baseados em Páginas	21
5.1 Granularidade	22
5.2 O Estudo do Caso ‘Ivy’	22
5.2.1 Implementação de Consistência Sequencial	23
5.2.2 Política de Substituição de Páginas	23
5.3 O Estudo do Caso ‘Mether’	23
5.3.1 Modelo de Consistência	24
6. Sistemas Baseados em Variáveis	24
6.1 O Estudo do Caso ‘Munin’	25
7. Sistemas Baseados em Objectos	25
7.1 O Estudo do Caso ‘Orca’	26
7.1.1 Gestão dos Objectos	26
7.2 Outros Sistemas	27
8. Discussão	27
9. Conclusões	29

10. Bibliografia **29**

1. Introdução

Com a actual redução dos custos associados ao fabrico de computadores, é cada vez mais atraente o aumento da velocidade de computação através da utilização de múltiplos processadores. Existem dois tipos básicos de sistemas (arquitecturas) para ligar vários processadores: os multiprocessadores onde os vários CPU's partilham blocos de memória física e os multicomputadores onde cada processador dispõe de um bloco de memória, não havendo qualquer partilha da mesma. Aos primeiros está normalmente associado o paradigma da comunicação por memória partilhada enquanto os segundos normalmente comunicam por troca de mensagens.

A programação utilizando troca de mensagens como modelo de comunicação é, regra geral, bastante mais complicada do que utilizando memória partilhada. No entanto, é bastante mais simples e mais barata a construção de multicomputadores do que de multiprocessadores. Por forma a combinar as vantagens dos multicomputadores com as vantagens do modelo de memória partilhada, surgiu um novo modelo que, implementado sobre o mecanismo de troca de mensagens, fornece a ilusão de memória partilhada. Este modelo é designado por Memória Partilhada Distribuída (DSM-Distributed Shared Memory) e é o tema principal deste trabalho.

Na secção seguinte será feita uma descrição comparativa entre os modelos de comunicação baseados em troca de mensagens e memória partilhada.

Na terceira secção serão apresentadas as arquitecturas multiprocessador e multicomputador, com uma breve descrição de alguns protocolos utilizados na gestão da memória partilhada dos multiprocessadores. Será também introduzido, nesta secção, o modelo de Memória Distribuída Partilhada .

Na quarta secção serão abordados alguns tópicos relacionados com a implementação da Memória Distribuída Partilhada.

Nas restantes secções serão descritas algumas implementações que utilizam DSM e será feita uma breve discussão/comparação entre elas.

2. Memória Partilhada vs Troca de Mensagens

Para permitir a programação de sistemas concorrentes existem dois paradigmas básicos no que respeita à comunicação e sincronização entre processos [LM92], [CDK94]:

- Troca de Mensagens (Message Passing 'MP');
- Memória Partilhada (Shared Memory 'SM').

A esquematização dos dois modelos pode ser vista na figura 1.

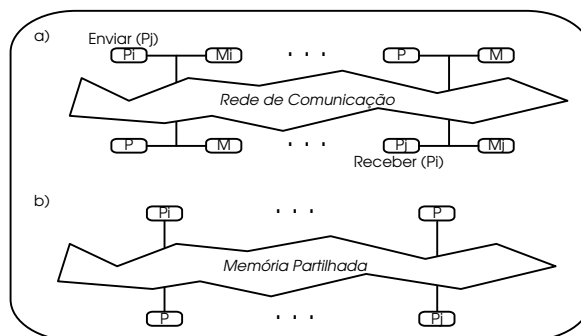


fig 1-paradigmas de programação concorrente. a)troca de mensagens, b)memória partilhada

Segundo o modelo de troca de mensagens, cada processo pode aceder aos dados no seu espaço de endereçamento utilizando as primitivas ler/escrever. Os processos comunicam entre si enviando e recebendo mensagens com as operações enviar/receber.

Quanto ao modelo de memória partilhada, todos os processos partilham o mesmo espaço de endereçamento. Assim, a comunicação é obtida através de operações de leitura e escrita na memória.

O paradigma de comunicação adoptado tem um grande impacto no modelo de programação. Utilizando o modelo de Troca de Mensagens, tem de existir, por parte do programador, uma plena consciência sobre a evolução do programa no que respeita ao fluxo de dados entre processos pois este é obtido explicitamente através das primitivas de envio e recepção. Todos estes mecanismos explícitos para obter a comunicação entre processos não são necessários quando se utiliza memória partilhada.

Uma outra desvantagem do modelo de troca de mensagens em relação ao modelo de memória partilhada está na dificuldade de enviar estruturas de dados pois estas têm de ser linearizadas e empacotadas antes de serem enviadas, o que representa um sobrecarga considerável. Por outro lado, e porque cada processo tem o seu espaço de endereçamento, é impossível a passagem de dados por referência, o que torna a passagem de estruturas complexas, como por exemplo grafos, extremamente complicada. O aparecimento do conceito de RPC (*Remote Procedure Call*) veio simplificar a comunicação com entidades remotas, não resolvendo, no entanto, o problema da passagem de dados por referência.

Outra diferença significativa entre os dois modelos reside no facto de ser obrigatório que, utilizando a troca de mensagens, a execução dos processos seja simultânea; utilizando memória partilhada e porque o espaço comum aos vários processos pode ser armazenado de uma forma persistente (ex. disco), um processo pode escrever os dados num endereço previamente acordado para serem posteriormente lidos por outro processo.

No que respeita às vantagens da comunicação por troca de mensagens, a mais significativa será talvez o maior isolamento entre os processos que este modelo permite: é muito mais fácil evitar que um processo em falha afecte todos os outros, como poderia acontecer utilizando memória partilhada (por ex. um processo a escrever indiscriminadamente na memória partilhada).

Em geral, e ressalvando algumas aplicações específicas, o modelo de programação com comunicação por memória partilhada é o preferido pelas suas potencialidades e simplicidade de programação.

3. Multiprocessadores, Multicomputadores e Memória Partilhada

Podemos classificar os sistemas com múltiplos processadores de acordo com a sua arquitectura [Tan95]. Assim, temos por um lado a categoria dos multiprocessadores (fig 2a)) da qual fazem parte os sistemas onde dois ou mais CPU's partilham uma memória física comum. A outra categoria é a dos multicomputadores (fig 2b)), onde não existe memória física partilhada entre os processadores. Embora esta distinção não seja por vezes muito clara, nem tão pouco consensual, ela será dentro do possível a adoptada neste texto.

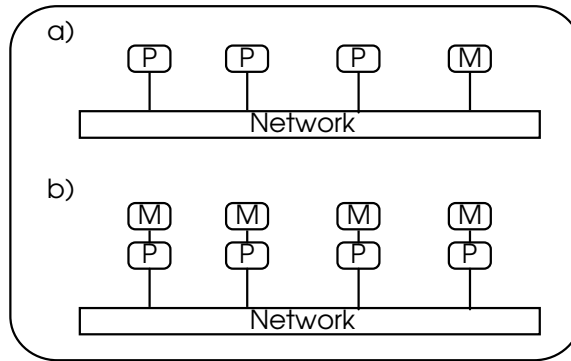


fig 2-arquitecturas com múltiplos processadores a) multiprocessador; b) multicomputador

Do ponto de vista do hardware, é extremamente difícil a construção de uma máquina onde um elevado número de processadores partilham uma mesma memória física. As duas arquiteturas mais comuns são a baseada em bus, que com o aumento do número de processadores tende a provocar a saturação do bus, e a ‘switched’ que permite a construção de grandes sistemas, sendo no entanto uma arquitetura extremamente complexa, lenta, de elevado custo e difícil manutenção (ver secções respectivas).

Por contraste, os multicomputadores são fáceis de construir em larga escala. É extremamente simples pegar num número praticamente ilimitado de computadores, contendo cada um, um CPU, memória e um interface de rede e ligá-los; ou seja, do ponto de vista do hardware os multicomputadores são preferidos aos multiprocessadores.

No que respeita ao software, o modelo utilizado para a programação dos multiprocessadores é o de memória partilhada (ver secção anterior) por ser directamente suportado pelo hardware. Quanto aos multicomputadores, devido às características da arquitectura, nomeadamente no que respeita à rede de ligação entre os computadores, a comunicação entre processos é baseada em troca de mensagens (ver secção anterior).

Resumindo, temos que os multiprocessadores são difíceis de construir e fáceis de programar, ao passo que os multicomputadores são fáceis de construir e difíceis de programar.

É neste ponto que surge o conceito de Memória Partilhada Distribuída (DSM - *Distributed Shared Memory*) como uma tentativa de desenvolvimento de um sistema fácil de construir como um multicomputador e fácil de programar como um multiprocessador.

Assim, temos a DSM como sendo uma abstracção usada para partilhar dados entre processadores que não partilham memória física. Esta abstracção é construída sobre arquiteturas multicomputador, utilizando o subsistema de troca de mensagens para, de uma forma transparente, fornecer os mecanismos necessários à implementação do modelo de programação baseado em memória partilhada. Esta abstracção pode ser esquematizada de acordo com a figura 3.

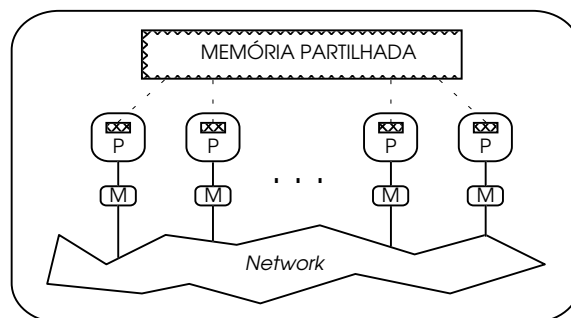


fig 3-modelo de memória partilhada distribuída (adaptado de [CDK94])

O resto desta secção será ocupado com uma breve descrição sobre o funcionamento dos multiprocessadores, especialmente a forma como implementam/gerem a memória partilhada, pois alguns dos algoritmos aplicados em DSM têm a sua origem neste tipo de arquitecturas.

3.1 Multiprocessadores Baseados em Bus

A forma mais simples de um multiprocessador baseado em bus consiste em alguns CPU's e um bloco de memória interligados por um bus (ver fig 4a). Neste tipo de sistemas, cada vez que um processador tem necessidade de recorrer a um valor da memória, fá-lo contactando directamente com a memória via bus. No entanto, este tipo de arquitectura simples tende a saturar o bus quando se aumenta o número de processadores (3-4 processadores é o limite). A solução mais comum reside na utilização de caches locais a cada processador, por forma a reduzir a comunicação com a memória central, aliviando o tráfego no bus (ver fig 4b).

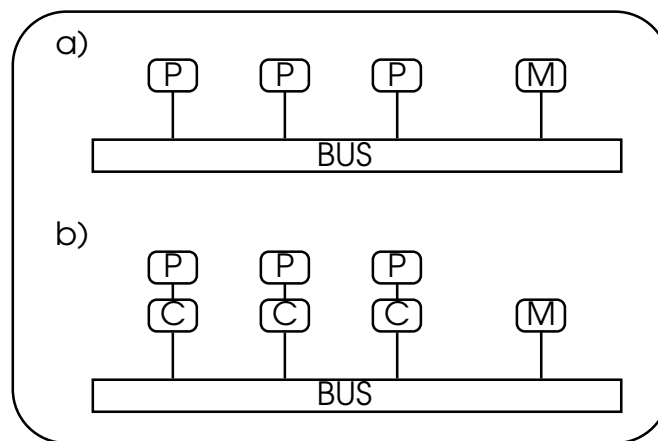


fig 4-multiprocessadores baseados em bus

A existência de caches múltiplas, num ambiente de memória partilhada, pode conduzir a inconsistências entre os dados, sendo necessário algum mecanismo para evitar esta situação. Uma forma de manter a consistência é obtida através da implementação do protocolo 'Snooping Cache' que será descrito em seguida.

3.1.1 Protocolo 'Snooping Cache'

Este protocolo assume a existência de um meio de comunicação que permita a difusão de informação, o que acontece em sistemas baseados em bus. Assim, cada cache tem de possuir um controlador que monitorize todas as transacções efectuadas no bus. Sempre que algum processador efectue uma operação de escrita, o endereço é colocado no bus para que todos os outros controladores de cache possam verificar se possuem alguma cópia do dado a ser alterado. Caso isto se verifique, existem várias alternativas das quais se destacam duas:

- os controladores de cache limitam-se a invalidar a cópia local, e da próxima vez que esse dado for requerido terá de ser novamente carregado da memória;
- após o envio do endereço, o controlador da cache responsável pela operação envia o novo valor do dado para que todas as caches que o contenham o possam alterar, mantendo assim a validade de todos os valores em cache.

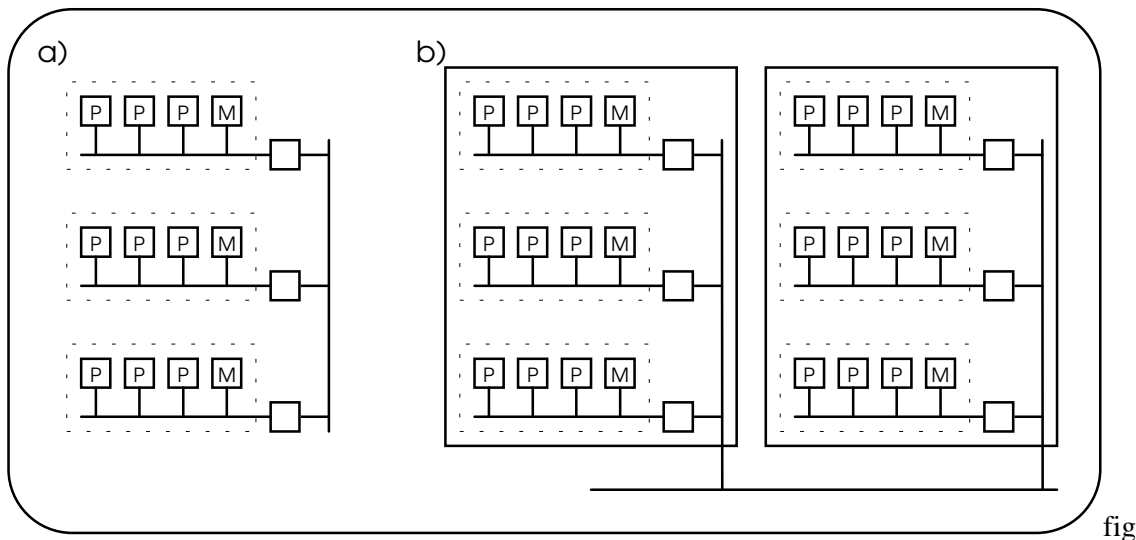
Existem outras variantes deste protocolo, utilizando no entanto o mesmo princípio básico.

3.2 Multiprocessadores 'Switched'

Apesar da introdução das caches permitir o aumento do número de processadores ligados a um único bus, haverá sempre um ponto em que o tráfego não permite o aumento de unidades

processadoras. Uma possível solução seria a introdução de múltiplos bus, aumentando a largura de banda. Outra solução passa pelo aumento da capacidade de comunicação utilizando para tal uma arquitectura com uma topologia diferente. Existindo várias propostas para a topologia, nomeadamente em forma de árvore ou grelha de bus. Estes sistemas são os chamados multiprocessadores ‘switched’, cujo princípio básico é sempre baseado numa hierarquia e é descrito em seguida:

ligar alguns CPU’s por um bus, só que agora este conjunto é visto como um *cluster* que será ligado por um bus *intercluster* a outros *clusters* de CPU’s. Se for necessário, podemos considerar um grupo de *clusters* como um *supercluster* e ligá-lo a outros *superclusters* por um bus adequado. Desde que os CPU’s comuniquem prioritariamente dentro do seu *cluster*, esta arquitectura é extremamente eficiente. A figura 5a) mostra vários *clusters* ligados por um bus *intercluster*, ao passo que a figura 5b) ilustra dois *superclusters* ligados por um bus *supercluster*.



5-a) três *clusters* ligados por um bus *intercluster* formando um *supercluster*; b) dois *superclusters* ligados por um bus *supercluster* (adaptado de [Tan95])

Neste tipo de sistemas, não é aplicável o protocolo ‘snooping cache’ para a manutenção da consistência devido à ineficiência da utilização da difusão. Assim, o protocolo utilizado é baseado em directorias e será descrito em seguida.

3.2.1 Protocolo Baseado em Directorias

Segundo este protocolo, é mantida informação sobre a localização e validade de todas as cópias da memória partilhada. Esta informação é mantida sob a forma de tabelas, em estruturas de hardware, que podem ser distribuídas ou centralizadas. Estas estruturas são as directorias. Assim, a consistência é mantida forçando as operações à memória partilhada a serem processadas pelo sistema de directorias. Este sistema executa as acções necessárias consoante o tipo de pedido e o estado do sistema. Existem várias versões deste protocolo, variando quanto ao número de cópias permitidas e informação mantida por bloco. No entanto o princípio básico reside no facto de não ser necessário um meio de comunicação com difusão pois as actualizações/invalidações podem ser enviadas sequencialmente apenas às caches que contêm os blocos em questão pois essa informação é mantida pelas directorias.

4. Aspectos Arquitecturais

Ao longo desta secção serão abordados vários aspectos relacionados com o projecto e implementação do modelo de DSM, como sejam os vários modelos de consistência, algoritmos básicos, estratégias de localização de dados, sincronização e estrutura dos dados.

4.1 Modelos de Consistência

Por forma a resolver o problema da ineficiência que pode representar a existência de uma única cópia de uma dada página, utiliza-se a replicação (múltiplas cópias) das mesmas. Este método introduz no entanto o problema relacionado com a manutenção da consistência entre todas as cópias. Manter uma consistência perfeita (todas as cópias rigorosamente iguais em cada instante) pode ser extremamente ineficiente devido à latência introduzida pela rede de comunicação [Mos93]. Uma possível solução seria enfraquecer o modelo de consistência, ou seja, permitir, nalgumas situações, estados transitórios de inconsistência. Apesar de resolver o problema da desempenho, a alteração do modelo de consistência adoptado introduz alterações no modelo de programação pois é necessário, por parte dos programadores, lidar com estas inconsistências transitórias. Podemos ver o modelo de consistência como sendo um contrato entre o software e a memória onde o software se compromete a obedecer a certas regras, sob pena de não haver garantia que o resultado das operações na memória produzam o resultado esperado. Em geral, um enfraquecimento no modelo de consistência introduz restrições e complexidade no modelo de programação. Assim, a escolha do modelo de consistência é um compromisso entre a desempenho da memória, a complexidade do modelo de programação e o próprio modelo de memória.

Podemos distinguir duas categorias genéricas de modelos de consistência:

- modelos uniformes: estes modelos não distinguem o tipo de acessos à memória, não necessitando portanto de variáveis de sincronização explícitas;
- modelos híbridos: nestes modelos, existe uma distinção entre os vários tipos de acesso à memória sendo necessária a utilização explícita de variáveis de sincronização.

No resto desta secção serão brevemente descritos vários modelos de consistência. Embora mencionando a maioria dos modelos actualmente utilizados, o desenvolvimento de novos modelos é assunto de investigação, não podendo portanto esta lista ser considerada como completa e fechada.

A descrição dos modelos será ilustrada com alguns exemplos que obedecem à seguinte notação:

- $R(x)a$ Leitura da célula x , retornando o valor a ;
- $W(y)b$ Escrita do valor b na célula y .

Serão representados ainda vários processos (P_1, P_2, \dots), que realizam as operações acima descritas segundo um eixo horizontal que representa o tempo (crescendo para o lado direito).

Ex:

$P_1:$	$W(x)1$
$P_2:$	$R(x)1$

No que respeita à notação utilizada, resta apenas referir que todas as variáveis são inicializadas a zero.

4.1.1 Consistência Atômica

É o mais forte de todos os modelos de consistência. Segundo este modelo, todas as operações ocorrem num dado instante de um tempo global. A necessidade da existência de um tempo global, que num sistema uniprocessador pode ser facilmente implementado, torna este modelo de consistência impossível de realizar num sistema distribuído.

Os seguintes exemplos mostram duas sequências de operações: (a) atomicamente consistente (b) não atomicamente consistente.

Ex:

a) $\frac{P_1: \quad W(x)1}{P_2: \quad \quad R(x)1}$	b) $\frac{P_1: \quad W(x)1}{P_2: \quad \quad R(x)0 \quad R(x)1}$
---	---

Seguindo o exemplo, o que a consistência atômica garante é que a escrita por parte de P₁ será visível à primeira leitura efectuada por P₂, por muito pequeno que seja o intervalo de tempo entre as duas operações.

4.1.2 Consistência Sequencial

Modelo de consistência definido por Lamport da seguinte forma:

“...o resultado de qualquer execução é o mesmo que se as operações de todos os processadores fossem executadas nalguma ordem sequencial, e as operações de cada processador individual aparecem nesta sequência na ordem especificada pelo seu programa.”

Esta definição significa que qualquer entrelaçamento das operações dos processadores é aceitável, desde que todos eles vejam a mesma sequência de referências à memória. No exemplo podemos ver dois possíveis resultados do mesmo programa.

Ex:

a) $\frac{P_1: \quad W(x)1}{P_2: \quad \quad R(x)0 \quad R(x)1}$	b) $\frac{P_1: \quad W(x)1}{P_2: \quad \quad R(x)1 \quad R(x)1}$
---	---

Refira-se a título de exemplo que a sequência de eventos do exemplo a) não respeita o modelo de consistência atômica.

Do ponto de vista do modelo de programação, este é um modelo atractivo, sendo inclusive um dos mais difundidos. No entanto, este modelo torna-se extremamente ineficiente em sistemas de larga escala, obrigando à investigação/implementação de outros modelos mais fracos.

4.1.3 Consistência Causal

Este modelo é uma versão mais relaxada da consistência sequencial. Segundo ele, apenas os eventos que são potencialmente causais (com uma relação causa/efeito) devem ser vistos na mesma ordem por todos os processadores. Os eventos concorrentes (não causais) podem ser vistos segundo ordens diferentes pelos vários processadores.

Na figura podemos ver um exemplo que obedece ao modelo de consistência causal, não sendo no entanto admissível segundo nenhum dos modelos anteriormente apresentados.

Ex:

$P_1: \quad W(x)1$	$W(x)3$	
$P_2: \quad \quad R(x)1 \quad W(x)2$		
$P_3: \quad \quad R(x)1$	$R(x)3 \quad R(x)2$	
$P_4: \quad \quad R(x)1$	$R(x)2 \quad R(x)3$	

No que respeita à implementação, este modelo implica a construção e manutenção de um grafo de dependências de operações, que envolve algum custo extra.

4.1.4 Consistência 'PRAM' (Pipelined RAM)

Segundo este modelo, mais fraco que o anterior, temos que:

- as operações de escrita feitas por um processador são vistas por todos os outros processadores na ordem pela qual foram executadas;
- as operações de escrita feitas por processadores diferentes podem ser vistas em ordens diferentes por processadores distintos.

O exemplo seguinte representa uma sequência de operações que obedece ao modelo de consistência 'PRAM', violando todos os modelos anteriormente apresentados.

Ex:

P ₁ :	W(x)1		
P ₂ :	R(x)1	W(x)2	
P ₃ :		R(x)1	R(x)2
P ₄ :		R(x)2	R(x)1

A vantagem deste modelo reside na sua fácil implementação. Considerando um sistema multiprocessador onde cada processador possui uma cópia local da memória partilhada, uma operação de leitura responderia sempre com o valor local; numa operação de escrita, o valor seria primeiro actualizado e posteriormente seria feito um difusão do novo valor para todos os outros processadores.

4.1.5 Consistência de Cache

Este modelo é uma versão relaxada da consistência sequencial, impondo apenas que os processadores concordem numa ordem no que respeita aos acessos a cada local.

O exemplo apresentado representa uma sequência que, não sendo sequencialmente consistente, obedece ao modelo de consistência de cache.

Ex:

P ₁ :	W(x)1	R(y)0
P ₂ :	W(y)1	R(x)1

Fazendo uma análise deste exemplo, temos que segundo o modelo sequencial nunca seria possível obter o valor zero nas duas operações de leitura pois qualquer entrelaçamento entre as operações dos dois processadores teria obrigatoriamente de começar com uma operação de escrita (W(x)1 ou W(y)1). No entanto, é possível, segundo o modelo de consistência de cache, serializar as operações referentes ao local x na forma R(x)0, W(x)1 e da mesma forma em relação a y poderíamos ter R(y)0, W(y)1.

4.1.6 Consistência de Processador

Este modelo representa uma combinação dos modelos 'PRAM' e de cache. Segundo este modelo, é necessário que todos os processadores concordem sobre a ordem de escritas de cada processador, podendo no entanto discordar na ordem de escritas de processadores diferentes desde que sejam referentes a locais na memória diferentes. O exemplo apresentado no modelo de consistência de cache é também válido segundo este modelo. No exemplo seguinte, encontra-se representada uma sequência de operações que **não** é admissível segundo o modelo de consistência de processador.

Ex:

P₁: $\frac{W(x)1 \quad W(c)1 \quad R(y)0}{W(y)1 \quad W(c)2 \quad R(x)0}$
 P₂: $\frac{W(y)1 \quad W(c)2 \quad R(x)0}{W(x)1 \quad W(c)1 \quad R(y)0}$

Neste exemplo, P₁ observa os acessos na seguinte ordem: W(x)1, W(c)1, R(y)0, W(y)1, W(c)2; enquanto P₂ observa a seguinte sequência de acessos: W(y)1, W(c)2, R(x)0, W(x)1, W(c)1, donde se pode ver que P₁ e P₂ discordam na sequência de escritas ao local 'c'.

4.1.7 Memória Lenta

Este modelo, sendo o mais fraco dos modelos uniformes, não tem grande aplicação prática. Fica apenas uma breve referencia por uma questão de completude.

Assim, segundo este modelo, todos os processadores concordam numa ordem no que se refere às escritas num dado local de memória feitas por um dado processador. Para além disso, as escritas têm que ser imediatamente visíveis localmente. O nome do modelo advém do facto das operações de escrita se propagarem muito lentamente.

A figura 6 representa a hierarquia de relações entre os modelos de consistência uniformes, do mais forte para o mais fraco.

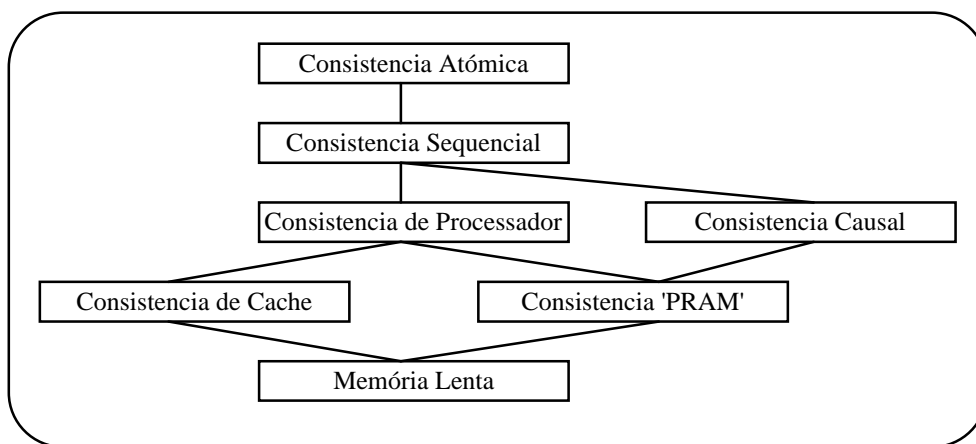


fig 6-hierarquia dos modelos de consistencia uniformes.

Apesar de se conseguir um ganho de desempenho através do enfraquecimento dos modelos de consistência apresentados, os modelos uniformes são em geral demasiado exigentes para as necessidades da maioria das aplicações. Na maior parte das vezes não é necessário que as escritas feitas por um processador sejam vistas pelos outros na sua correcta ordem, sendo que nalguns casos, estas escritas nem sequer requerem a actualização de todas as cópias de memória como por exemplo no caso em que um processador está dentro duma secção crítica a manipular em ciclo uma variável onde apenas é necessário propagar o resultado final. Por forma a utilizar certas propriedades, conhecidas à priori, das aplicações, no melhoramento da desempenho, foram definidos vários modelos de consistência que têm em comum o facto de distinguirem os vários tipos de acessos à memória, através da introdução de variáveis de sincronização. Estes modelos são designados por modelos híbridos e serão descritos de seguida.

4.1.8 Consistência Fraca

Este modelo, sendo o mais forte dos modelos híbridos, é descrito pelas seguintes propriedades:

- o acesso às variáveis de sincronização é sequencialmente consistente;
- nenhum acesso a uma variável de sincronização é permitido até que todos os acessos anteriores tenham sido completados;
- nenhum acesso é permitido até que todos os acessos a variáveis de sincronização anteriores tenham sido completados.

Ou seja, os dados partilhados apenas podem ser considerados como consistentes após uma sincronização.

O exemplo seguinte representa uma sequência válida segundo o modelo de consistência fraca.

Ex:

P ₁ :	W(x)1	W(x)2	S
P ₂ :		R(x)1	R(x)2 S
P ₃ :		R(x)2	R(x)1 S

onde S representa a sincronização.

4.1.9 Consistência 'Release'

Segundo o modelo anterior, sempre que havia um acesso a uma variável de sincronização, não havia forma de saber se o processo acabara de escrever na memória ou se iria começar a lê-la, sendo portanto necessário executar as acções requeridas por ambas. Uma forma de contornar este facto, melhorando a desempenho, é a introdução de dois tipos de operações de sincronização:

- *Acquire*: usada para informar o sistema de memória que se vai entrar numa região crítica;
- *Release*: usada quando se acabou de sair de uma região crítica.

Cabe ao programador, segundo este modelo, a responsabilidade de incluir as linhas de código necessárias, no programa, para informar o sistema de memória sobre estes eventos. Assim, o sistema de memória, em resposta a um *Acquire*, garante a actualização das variáveis protegidas. Depois de um *Release*, as variáveis protegidas que tenham sido alteradas são propagadas para os outros processadores.

O exemplo seguinte representa uma sequência de eventos válida segundo este modelo:

Ex:

P ₁ :	Acq(L)	W(x)1	W(x)2	Rel(L)
P ₂ :		Acq(L)	R(x)2	Rel(L)
P ₃ :				R(x)1

Uma possível optimização deste modelo, por forma a economizar largura de banda, foi implementada sob a designação de '*lazy release consistency*'. Segundo este algoritmo, as alterações são propagadas apenas quando requeridas por um processador tentando fazer um *Acquire*. O ganho obtido pode ser ilustrado com o exemplo de um programa com uma zona crítica dentro de um ciclo. Com consistência *Release* normal, as alterações teriam de ser propagadas sempre que se saia da zona crítica. Com o algoritmo *lazy*, é grande a probabilidade de não ser necessário propagar nenhum valor durante todo o ciclo.

4.1.10 Consistência de Entrada

É um modelo híbrido que por ser mais fraco que o anterior impõe mais restrições ao modelo de programação. Segundo este modelo, cada variável partilhada é associada a uma variável de sincronização. A vantagem introduzida por esta alteração reside no facto de ser agora possível um acesso concorrente a diferentes secções críticas. É necessário, no entanto, ao nível da programação, declarar as associações entre as variáveis partilhadas e de sincronização. Segundo este modelo, existem dois tipos de acessos: exclusivo e não exclusivo, aumentando o processamento concorrente.

4.2 Algoritmos Básicos

Nesta secção serão descritos quatro algoritmos básicos que implementam memória partilhada distribuída [SZ90]. Estes algoritmos podem ser classificados de acordo com a utilização de migração e/ou replicação (ver fig 7).

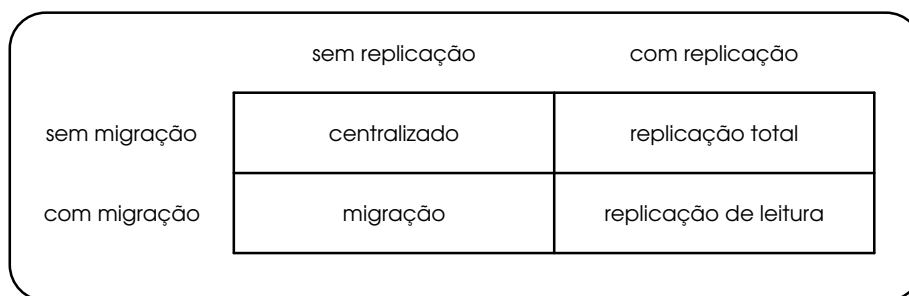


fig 7-algoritmos básicos (adaptado de [SZ90])

A replicação pode reduzir os custos médios das leituras porque permite uma maior concorrência no acesso aos dados com menor utilização da rede. No entanto, as operações de escrita podem ter maiores custos, pelo facto de ser necessário invalidar ou actualizar todas as outras réplicas por forma a manter um estado consistente. A vantagem da replicação é proporcional à relação entre leituras e escritas. As implementações da DSM baseadas em replicação devem torná-la transparente para a aplicação. Os processos não devem saber se os dados são sempre lidos da mesma cópia ou não.

4.2.1 Modelo centralizado

A forma mais simples de implementar a DSM consiste na utilização de um servidor central, responsável por coordenar e gerir toda a memória partilhada. É a este servidor que todos os pedidos relacionados com os dados contidos na DSM devem ser dirigidos pois ele gere a única cópia existente dos dados. Assim, por cada acesso, é enviada uma mensagem ao servidor, respondendo este com outra mensagem contendo os dados requeridos no caso de uma leitura, ou um *acknowledge* no caso de uma operação de escrita (ver fig 8). O custo associado a este protocolo é assim de duas mensagens na rede. Por forma a implementar este protocolo, é apenas necessário um modelo de comunicação simples do tipo request/reply utilizando os mecanismos habituais de controle de falhas (timeout, detecção de duplicados, limitação do número de retransmissões).

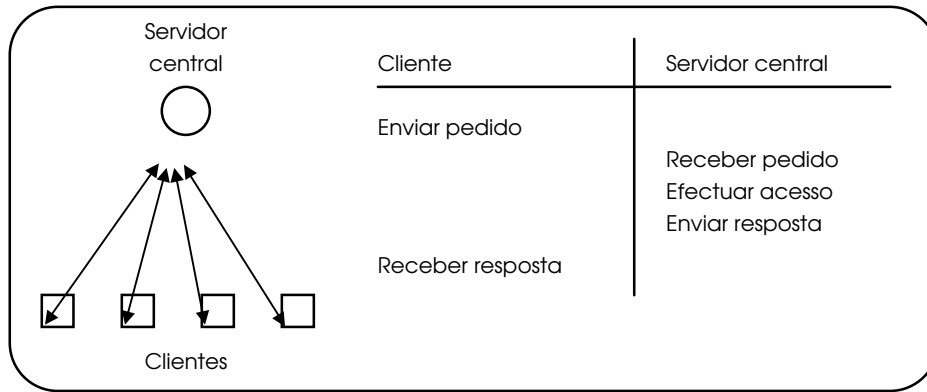


fig 8-modelo centralizado (adaptado de [SZ90])

O facto de todos os pedidos terem de ser processados por uma única entidade, torna este modelo impraticável num sistema com um elevado número de acessos à memória, devido ao congestionamento que produziria no servidor central. Uma possível solução para este problema passa pela utilização de vários servidores, gerindo cada um uma parcela do espaço de memória. Esta solução introduz, no entanto, a necessidade de localizar o servidor responsável por um dado bloco de memória, por parte dos clientes do serviço.

4.2.2 Modelo de migração

Segundo este modelo, existe apenas uma cópia dos dados existentes na memória partilhada, podendo esta migrar para o local que dela necessita. Os processos apenas podem aceder aos dados locais. A migração é feita em unidades de tamanho fixo chamadas bloco. Este modelo tira partido da localidade de referências apresentada pela maioria dos programas devido ao facto de não haver qualquer comunicação envolvida no acesso a uma página local. Um ponto muito sensível deste algoritmo está relacionado com o tamanho dos blocos (ver secção sobre Granularidade no capítulo sobre Sistemas Baseados em Páginas).

Uma segunda vantagem deste modelo é a de poder ser integrado com o sistema de memória virtual, caso o tamanho de bloco seja igual ao tamanho da página de memória virtual. Os dados que estiverem locais podem ser projectados no espaço de endereçamento virtual da aplicação e serem acedidos através das primitivas de sistema. Sempre que um dado migra, é retirado de todos os espaços de endereçamento locais em que tenha sido projectado. Caso um dado bloco acedido não resida localmente, é desencadeada uma falta de página fazendo com que o gestor de faltas comunique com os seus pares para obter o referido bloco (ver fig 9). Uma desvantagem deste modelo, reside no facto de os dados apenas poderem ser acedidos pelas entidades locais.

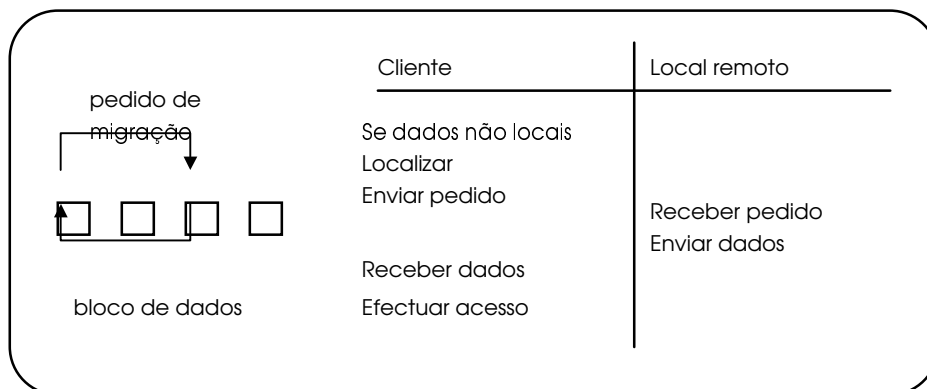


fig 9-modelo de migração (adaptado de [SZ90])

4.2.3 Modelo de replicação de leitura

Para resolver a ineficiência que representa a existência de apenas uma cópia de cada dado, introduz-se o conceito de replicação (várias cópias).

Este modelo, utilizando a replicação como forma de melhorar a desempenho, distingue entre dois tipos de cópias: de leitura e de escrita. Assim, temos a possibilidade de coexistência de várias cópias (réplicas) de leitura, ao passo que é apenas permitida a existência de uma cópia de escrita.

Segundo este modelo, existe o conceito de dono de uma cópia, como sendo o local que detém a única cópia com permissões de escrita. Existem várias hipóteses de implementação deste modelo, variando no que respeita ao dono da página com permissões de escrita, bem como no que respeita ao modelo de consistência adoptado. Assumindo que o dono da página não é fixo, o princípio básico é, o seguinte:

no caso de um determinado processo que deseja fazer uma operação de escrita não ser o dono da página em questão, ele tem que entrar em contacto com o dono da cópia por forma a obter as referidas permissões bem como uma cópia válida da página caso esta não exista localmente. É igualmente necessária a invalidação de todas as cópias existentes no sistema. Só após este processo, pode a operação de escrita ser concretizada (ver fig 10), passando este local a ser o dono da cópia. Este processo é normalmente designado por 'write-invalidate'.

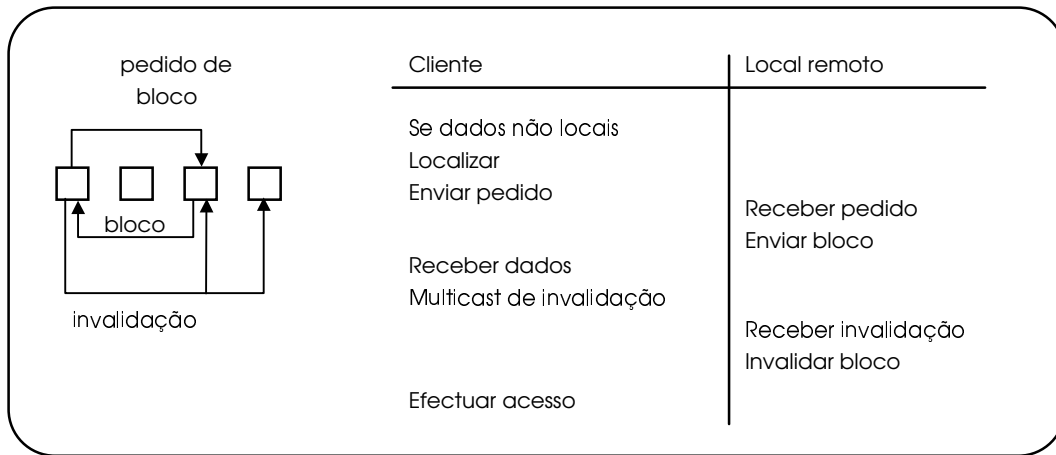


fig 10-operação de escrita no modelo de replicação de leitura (adaptado de [SZ90])

No caso de ser desejada uma operação de leitura, é necessário obter uma cópia de leitura, caso esta não exista localmente.

4.2.4 Modelo de replicação total

Segundo este modelo, são permitidas réplicas dos dados mesmo no caso em que estes estão a ser alterados. No modelo anterior, era bastante simples a manutenção de um estado consistente devido ao facto de todas as operações de escrita serem sequenciadas de acordo com a ordem pela qual são feitas, no local que detém os dados. Com a introdução do conceito de replicação, a manutenção de um estado consistente tem, obrigatoriamente, de passar por uma correcta sequencialização ou controle dos acessos aos dados.

Existindo várias formas de implementação deste tipo de protocolo, consoante o modelo de consistência desejado, uma delas, que implementa a consistência sequencial, reside na existência de um sequenciador global, para o qual são dirigidos todos os pedidos de escrita. Este sequenciador atribui um número de ordem a cada um dos pedidos, fazendo em seguida um difusão para todos os locais que contenham uma cópia dos dados (ver fig 11). Este processo é conhecido por 'write-update'. Uma alteração a este algoritmo consiste em não enviar mensagens de alteração para todos os locais. Assim, o sequenciador ao receber um pedido de

escrita, introdu-lo num registo, e é enviada uma confirmação ao processo que o enviou. As actualizações apenas serão enviadas para e quando um dado local enviar um pedido de escrita.

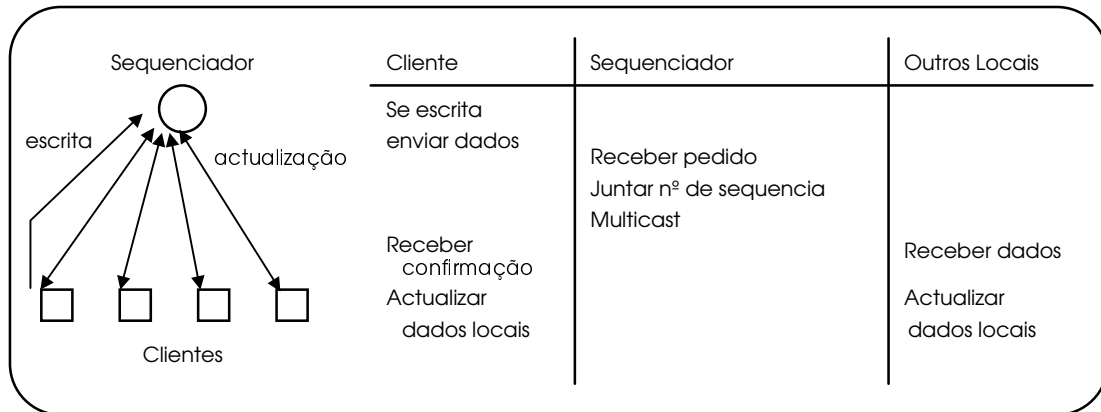


fig 11-operação de escrita no modelo de replicação de total (adaptado de [SZ90])

4.3 *Dono, Réplicas e sua Localização*

No que respeita à posse das páginas, ela pode ser fixa ou dinâmica [LH89].

Segundo a abordagem fixa, o dono da página é sempre o mesmo processador. Aos outros processadores nunca são dados plenos direitos de escrita sobre essa página, tendo estes que gerar uma ‘falta de escrita’ cada vez que necessitam de fazer uma actualização sobre a mesma. Este facto torna esta situação impraticável.

Quanto à abordagem dinâmica, podemos ter uma gestão centralizada ou distribuída, podendo esta última ser subdividida numa abordagem fixa ou dinâmica.

Segue-se uma descrição mais detalhada das três abordagens nas quais o dono da página não é fixo.

4.3.1 *Gestão Centralizada*

Segundo este tipo de modelo existe um gestor centralizado, residente num único processador que mantém uma tabela (directoria) com entradas relativas a todos os blocos de memória contendo, cada uma, informação sobre:

O dono da página ou seja o último processador a ter direitos de escrita .

A lista de todos os processadores que possuem réplicas, por forma a que as alterações/invalidações possam ser feitas sem recurso a difusão.

Cada processador localmente mantém uma tabela com informação sobre as páginas que detem.

Assim, sempre que um processo tem a necessidade de obter uma cópia, contacta o gestor central. Este, após determinar qual o dono da página, consultando a tabela que mantém, pode processar o pedido de acordo com uma das seguintes alternativas:

- Responder com o endereço do dono da página, ficando o processador inicial responsável por efectuar o contacto directo com o dono da página (ver fig 12a), necessitando ao todo de quatro mensagens na rede;
- Contactar directamente com o dono da página, dando-lhe indicação do pedido, ao qual este deve responder com o envio da página ao processador inicial (ver fig 12b), necessitando para tal um total de três mensagens na rede.

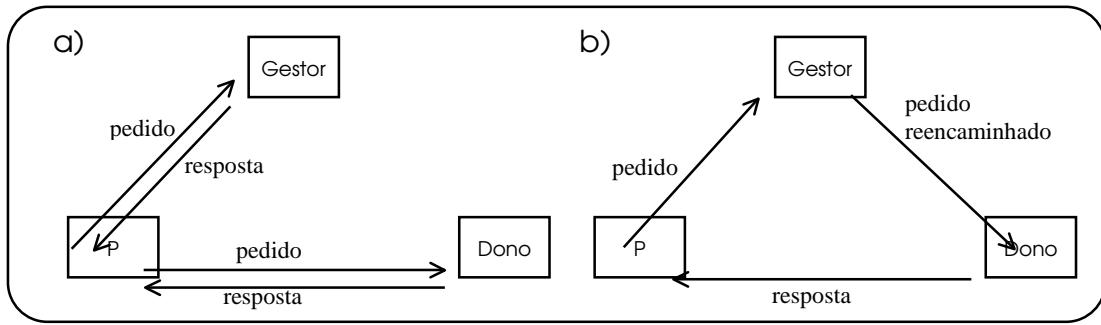


fig12 - a)solução recursiva b)solução transitiva (adaptado de [Tan95])

4.3.2 Gestão Distribuída Fixa

O facto de toda a gestão da DSM ter obrigatoriamente de passar por um único gestor torna a abordagem centralizada de alguma forma ineficiente, podendo sobrecarregar o referido gestor.

Surge assim a versão distribuída fixa como sendo uma extensão do modelo anterior. Assim, em vez de um único gestor, passam a coexistir vários gestores, sendo cada um responsável por um determinado conjunto de páginas. O problema que surge com este tipo de abordagem está relacionado com o modo como se faz a distribuição das páginas pelos gestores, e mais particularmente com a forma como um dado processador determina qual o gestor de uma determinada página. A solução deste problema passa pela correcta escolha de uma função de mapeamento entre as páginas e os gestores.

4.3.3 Gestão Distribuída Dinâmica

Este modelo elimina o conceito de um gestor separado como entidade que mantém a directoria de todas/algumas páginas, transferindo para os processadores individuais a responsabilidade de localização do dono de cada página. Assim, cada processador mantém localmente uma directoria, contendo informação sobre todas as páginas do sistema, onde, em vez do campo correspondente ao dono da página, existe um campo referente ao dono provável. Este campo contém apenas uma pista relativamente a quem o dono possa ser.

Assim, sempre que é desencadeada uma ‘falta de página’, é enviado um pedido ao processador considerado como sendo o dono provável. Caso ele seja o dono, o procedimento decorre como no modelo centralizado. Caso contrário, este processador remete o pedido para o processador referenciado no seu campo relativo ao dono provável, alterando o campo para o endereço do processador que gerou o pedido (ver fig 13).

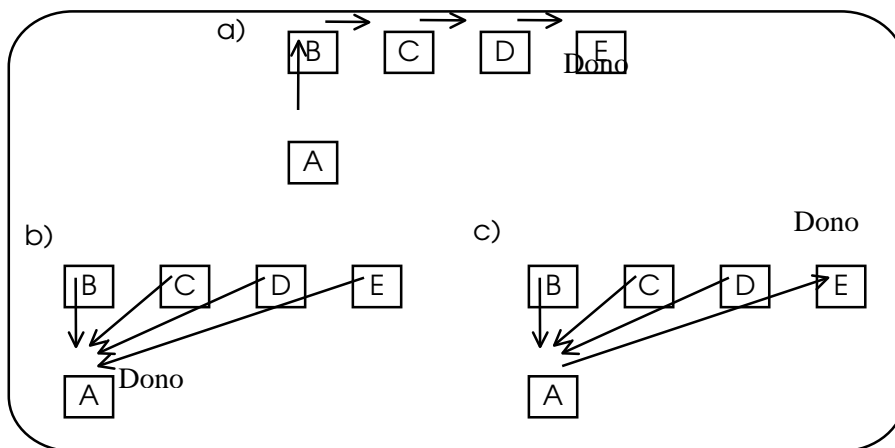


fig 13 -Ponteiros para dono provável a)antes de ‘A’ gerar a falta de página b)após ‘A’ gerar uma falta de escrita c)após ‘A’ gerar uma falta de leitura (adaptado de [CDK94])

4.4 Sincronização

A implementação de um protocolo de consistência, por si só, não garante a correcta serialização do acesso concorrente a variáveis partilhadas [CDK94].

Vamos supor, por exemplo, que a e b são duas variáveis partilhadas sujeitas à restrição $a=b$. Supondo ainda que dois ou mais processos executam o seguinte código:

```
a:=a+1;
```

```
b:=b+1;
```

pode, neste caso, surgir um estado inconsistente. Assim, se a e b forem inicializadas a zero, e o processo 1 incrementar a variável a , pode ocorrer a situação em que antes que o processo possa incrementar a variável b , um processo 2 incrementa ambas as variáveis, introduzindo o estado $a=2, b=1$, inconsistente com a restrição imposta.

Para lidar com este tipo de situações são necessários mecanismos através dos quais os processos possam aceder aos dados partilhados de uma forma coordenada.

No resto desta secção serão brevemente descritos alguns destes mecanismos existentes, e será feita uma muito breve análise da forma como podem ser implementados [RS93].

4.4.1 Mecanismos Básicos de Sincronização

4.4.1.1 Semáforos

Introduzidos para resolver, entre outros, o problema da exclusão mutua, um semáforo é uma variável de sincronização que é iniciada a um (semáforo binário) ou a um número superior (semáforo de contagem de recursos).

São definidas as seguintes duas operações (P e V) sobre um semáforo:

<pre>P(s): if (S ≥ 1) then s:=s-1; else bloquear o processo na fila do semáforo</pre>	<pre>V(s): if (fila não está vazia) then retirar um processo da fila else s:=s+1;</pre>
---	---

4.4.1.2 'Eventcounts'

Um *eventcount* é uma variável inteira, monotonicamente crescente. Sempre que um processo necessita de sinalizar um dado evento a outro processo, fá-lo incrementando o *eventcount*.

Estão definidas as seguintes operações sobre este tipo de variáveis:

- *advance(eventcount)*: incrementa o *eventcount*;
- *read(eventcount)*: lê o valor do *eventcount*;
- *await(eventcount, valor)* bloqueia o processo até *eventcount* atingir o 'valor' especificado.

Este mecanismo foi proposto como alternativa para resolver o problema da exclusão mútua.

4.4.1.3 'Locks'

Neste tipo de mecanismo, cada recurso partilhado é associado a um *lock*. Um dado processo obtém acesso exclusivo a um dado recurso adquirindo o *lock* respectivo.

As duas operações definidas são:

- *acquire(lock)* : obter o *lock*;

- `release(lock)`: libertar o *lock*.

4.4.1.4 Barreiras

Mecanismo existente para permitir a sincronização de vários processos em certos pontos durante a execução do programa. As barreiras tornam-se especialmente úteis quando é por exemplo necessário que todos os processos completem uma dada fase de computação antes de avançarem para a secção seguinte. Quando um processo atinge uma barreira, ele é bloqueado até que todos os outros processos aí cheguem. Nessa altura, todos os processos são desbloqueados.

4.4.2 Implementação

A implementação dos mecanismos de sincronização pode ser classificada de acordo com os seguintes critérios:

4.4.2.1 Hardware vs Software

Em geral esta distinção está fortemente relacionada com a classificação do sistema como sendo um multiprocessador ou um multicomputador. Nos sistemas multicomputadores, alvo preferencial deste trabalho, onde em geral não existe um suporte de hardware para a partilha de memória, os mecanismos de sincronização são implementados em software sob a forma de bibliotecas ou mesmo integrados no sistema de operação reduzindo o sobrecarga associado. Nos multiprocessadores, a solução adoptada para a implementação dos mecanismos de sincronização é em geral de hardware.

4.4.2.2 Centralizado vs Distribuído

Num sistema centralizado, um nó particular é definido como sendo o gestor central, recebendo e processando todos os pedidos de sincronização. É a forma mais fácil de implementar, tendo ainda como vantagem o facto de não ser necessário localizar o dono da variável de sincronização. Tem como sérias desvantagens por um lado o facto de se poder tornar num 'bottleneck' e por outro o facto de ser um ponto muito sensível a falhas. Uma alteração a este esquema, que melhora os seus pontos negativos, reside na utilização de mais do que um gestor por sistema, onde cada um processa um subgrupo das variáveis de sincronização. Esta abordagem é designada por distribuída estática. Apesar de ser melhor que a anterior, esta solução continua a ser centralizada, podendo o seu desempenho decair no caso de frequentes acessos a variáveis processadas pelo mesmo gestor.

Numa solução distribuída dinâmica, as variáveis de sincronização podem migrar entre nós, sendo o dono de cada uma, em cada momento, o nó que a possui. Esta solução elimina os problemas encontrados nas soluções estáticas, introduzindo no entanto o problema da localização de uma dada variável de sincronização. Este problema pode ser resolvido utilizando técnicas em tudo semelhantes às utilizadas para a localização de páginas de memória, anteriormente descritas.

4.4.2.3 Integrado vs Não Integrado

Um mecanismo de sincronização é integrado com o sistema de DSM se é implementado da mesma forma que os outros dados da memória partilhada. Um sistema DSM baseado em páginas, onde por exemplo semáforos sejam implementados em páginas partilhadas de semáforos (com uma possível semântica diferente) é um caso de um sistema integrado.

Um sistema não integrado é aquele cuja implementação não é influenciada pelo sistema de DSM. Podemos considerar o exemplo acima descrito, mas desta vez com os semáforos implementados numa forma estática.

4.5 Estrutura do Espaço Partilhado

Os sistemas que implementam o paradigma da Memória Partilhada Distribuída podem ser classificados de acordo com a estrutura do espaço partilhado como sendo [Tan95]:

- baseados em páginas;
- baseados em variáveis;
- baseados em objectos.

Os sistemas baseados em páginas (abordagem clássica também conhecida por Memória Virtual Distribuída), têm por objectivo a simulação de uma memória física partilhada real, tendo para isso a página, como unidade de partilha. A ideia principal associada a estes sistemas era a de permitir executar os programas escritos para multiprocessadores, permitindo assim um melhor desempenho.

Por forma a resolver alguns problemas existentes nos sistemas baseados em páginas entre os quais o da falsa partilha (problema que surge quando duas variáveis independentes, frequentemente usadas por processadores diferentes, residem na mesma página), apareceram os sistemas baseados em variáveis, que, adoptando uma abordagem mais estruturada, permitem a partilha de apenas certas variáveis e estruturas de dados que sejam necessários a mais do que um processador.

Por último temos os sistemas baseados em objectos como sendo a abordagem mais estruturada. Segundo este modelo, a unidade de partilha é o objecto. Como motivação deste tipo de sistemas está o cada vez maior sucesso da programação orientada por objectos. Dado que segundo este modelo, quando se acede a parte de um objecto, na maioria dos casos todo o objecto é necessário, faz todo o sentido que os dados se movam em unidades de objecto em vez de unidades de páginas. Por outro lado, e dada a obrigatoriedade de os objectos serem acedidos através dos métodos para eles definidos, obtém-se assim uma forma implícita de obter a exclusão mútua bem como um mecanismo mais forte de protecção a erros.

Uma descrição/comparação mais detalhada sobre este assunto será feita quando forem apresentadas algumas implementações destes modelos.

5. Sistemas Baseados em Páginas

Este tipo de sistemas são aqueles que implementam o modelo clássico de memória partilhada distribuída. O seu objectivo é fornecer aos programadores um espaço de endereçamento que possa permitir a partilha de qualquer tipo de dados, entre processos que residam em locais distintos, de uma forma transparente.

Sendo a primeira abordagem no que respeita à implementação de DSM, o seu principal objectivo era o de permitir a execução de programas desenvolvidos para sistemas multiprocessadores, sem que para isso houvesse necessidade de efectuar alterações.

O princípio básico inerente a estes sistemas era o de tentar simular a cache dos multiprocessadores com a memória local, através de software do sistema de operação e das Unidades de Gestão de Memória. Assim, o espaço de endereçamento partilhado era dividido em blocos de tamanho fixo que eram distribuídos por todos os processadores do sistema. Quando um processador faz uma referência a um bloco não residente, ocorre uma falta e o software da DSM encarrega-se de ‘resolver’ o assunto. Várias formas de obter este comportamento foram descritas no capítulo anterior.

Antes, no entanto, será abordado um assunto bastante importante para um bom desempenho do sistema: o tamanho dos blocos ou seja a granularidade.

5.1 Granularidade

Ao contrário do que acontece nos sistemas multiprocessadores, cujas unidades de transferência de dados entre processadores são bastante pequenas (alguns bytes) devido à velocidade do meio de comunicação, em sistemas distribuídos isto é impraticável devido à sobrecarga introduzida pelas redes de comunicação. Uma solução bastante óbvia seria adoptar como unidade de transferência a página, com o benefício extra de permitir uma integração do sistema de DSM com a Unidade de Gestão de Memória [Tan95]. Aumentar ainda mais a unidade de transferência, para um número múltiplo do tamanho da página, teria como vantagem o facto de globalmente reduzir a sobrecarga nas comunicações (é mais eficiente transmitir um bloco de 1024k do que 2 blocos de 512k cada), sabendo que, devido à propriedade da localidade de referência que a maioria dos programas apresenta, esses dados seriam todos utilizados.

No entanto, adoptar para unidade de transmissão blocos de tamanhos muito elevados tem as suas desvantagens.

Considere-se o caso da figura 14 onde existe uma página de memória partilhada contendo duas variáveis, cada uma a ser utilizada por um processador. Neste caso, a desempenho do sistema iria decair significativamente devido ao facto da página contendo as duas variáveis ter de ser constantemente transmitida entre os processadores quando na verdade eles não partilham quaisquer dados. Este problema é designado por ‘falsa partilha’ e surge num grau proporcional ao tamanho da unidade de transferência.

Daqui se pode ver a importância que a granularidade dos dados representa no bom desempenho dum sistema. Unidades de transferência na ordem dos 512 bytes são comuns.

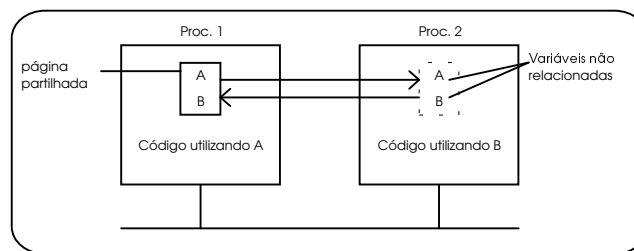


fig 14-Falsa partilha (adaptado de [Tan95])

5.2 O Estudo do Caso ‘Ivy’

É talvez importante começar por referir a importância histórica deste sistema no contexto da DSM. Assim, foi não só um dos primeiros sistemas a implementar este modelo, como também um dos estudos mais exaustivos sobre os vários algoritmos de gestão de DSM. Uma grande parte dos assuntos descritos no capítulo anterior foram inicialmente abordados por este sistema, com particular relevo para os vários algoritmos relacionados com a gestão e detenção das páginas.

Este sistema [LH89], implementado sobre o Apollo Domain, oferece a cada processo um espaço de endereçamento contendo duas partes distintas: uma privada, à qual apenas esse processo pode aceder, e uma parte partilhada por vários processos.

O Ivy adopta como unidade de transferência a página, integrando desta forma a gestão da DSM com a Unidade de Gestão de Memória (MMU) local. São oferecidos ‘locks’ e ‘eventcouts’ como primitivas de sincronização, sendo estas geridas de uma forma integrada com o sistema de DSM.

Utilizando um protocolo de ‘write-invalidate’, segue-se uma breve descrição sobre a forma como é mantida a consistência do sistema.

5.2.1 Implementação de Consistência Sequencial

Utilizando um modelo de replicação de leitura, sendo considerado o dono de uma página o último processador a efectuar uma operação de escrita, são implementados vários protocolos de consistência, todos eles utilizando o conceito de directoria introduzido pelos multiprocessadores ‘switched’ [TSF90].

Cada MMU contém uma tabela de páginas onde está incluída informação sobre o estado das mesmas, numa forma em tudo semelhante a uma cache. Assim cada página pode estar num dos seguintes estados: ‘write’, ‘owned read’, ‘read’, ‘nil’ ou ‘unused’. Paralelamente, o dono de uma página (ou o gestor da mesma, caso seja uma entidade distinta) mantém uma lista de cópias contendo referências sobre todos os processadores que possuem réplicas da mesma.

Sempre que é efectuada uma referência a uma página não existente localmente com as permissões adequadas, é gerada uma falta de página que é tratada da seguinte forma:

Falta de leitura:

- Localizar o dono da página;
- Pedir uma cópia ao dono;
- Acrescentar uma referência sobre a nova réplica na lista de cópias;
- O dono da página altera o estado da sua página para ‘owned read’;
- O dono envia a página;

Falta de escrita:

- Localizar o dono da página;
- Pedir uma cópia com direitos de escrita;
- O dono envia a página e a lista de cópias, alterando o estado da sua cópia para ‘nil’;
- O novo dono envia mensagens de invalidação a todos os processadores que constarem na lista de cópias (estas invalidações são enviadas pelo gestor da página, caso exista um);
- Após a recepção de todas as confirmações a escrita pode ser efectuada.

No que se refere à localização do dono das páginas, foram implementados os vários modelos descritos no capítulo anterior, nomeadamente o modelo de gestão centralizada, distribuída fixa e distribuída dinâmica.

5.2.2 Política de Substituição de Páginas

Devido ao facto do espaço de memória numa máquina ser sempre limitado e normalmente muito menor do que o espaço de endereçamento virtual, é necessário implementar algum esquema de substituição de páginas.

No Ivy, sempre que é necessário utilizar uma nova página, as primeiras a ser utilizadas são as que estão no estado ‘nil’ ou ‘unused’. Caso não exista nenhuma página num destes estados, serão substituídas as páginas com estado ‘read’.

No caso de ser necessário substituir páginas no estado ‘write’ ou ‘read owned’, elas serão armazenadas em disco ou, alternativamente, será encontrado um outro processador que contenha uma cópia válida da página em questão, o qual passará a ser o novo dono da página, podendo então a cópia local ser substituída.

5.3 O Estudo do Caso ‘Mether’

Manter um estado de consistência sequencial pode ser bastante penalizante, devido ao grande número de troca de mensagens necessário, e conseqüente elevada latência da rede.

O enfraquecimento do modelo de consistência, embora à custa de alterações no modelo de programação, leva a uma redução no número de mensagens na rede, diminuindo a latência introduzida pela mesma. Alterar a semântica da consistência pode levar a três grandes vantagens:

- melhorar o desempenho do sistema;
- permitir um maior crescimento do sistema;
- aumentar a tolerância a mensagens perdidas.

Assim, o sistema Mether [MF89] implementa o modelo de consistência fraca (ver sec. correspondente), ou seja, um dado processador pode fazer várias alterações numa dada página sem que essas sejam propagadas para as restantes cópias. Paralelamente, e por forma permitir a execução de aplicações que exijam um modelo de consistência sequencial, são fornecidos mecanismos próprios para que seja possível aceder a uma cópia consistente segundo este modelo.

Além de facultar ao programador estes dois modelos de consistência, o Mether permite ainda a escolha do tamanho da página de entre dois possíveis: 32 e 8192 bytes, bem como o facto das operações serem bloqueantes (o processo fica bloqueado sempre que é desencadeada uma falta de página) ou não.

5.3.1 Modelo de Consistência

Um programa pode escolher entre um acesso a:

- uma página sequencialmente consistente, caso em que esta será a única cópia com permissões de escrita;
- uma página inconsistente, caso em que a cópia que lhe será fornecida é uma réplica da cópia consistente.

Ao longo do tempo, as cópias inconsistentes são actualizadas quer por vontade expressa do seu dono, quer por imposição do dono da cópia consistente. Existe, no entanto, um mecanismo interno que garante a não existência de cópias desactualizadas, por mais de 30 segundos.

A transmissão de páginas na rede é feita recorrendo à utilização de um ‘kernel driver’, sendo a consistência mantida por um servidor situado ao nível do utilizador.

6. Sistemas Baseados em Variáveis

Os sistemas descritos no capítulo anterior, baseados em páginas, têm por objectivo proporcionar um espaço de endereçamento linear, partilhado por vários processos, por forma a permitirem a execução dos programas escritos especialmente para os multiprocessadores. Analisando-se a forma como normalmente os programas utilizam a memória partilhada para comunicarem, conclui-se que apenas algumas variáveis são utilizadas em comum. Este facto levou ao desenvolvimento de uma forma mais estruturada de implementação de DSM, na qual não existe a noção de um espaço linear partilhado, mas sim, um conjunto de variáveis/estruturas partilhadas. À partida, e proque a unidade de partilha é a variável, é possível eliminar o problema da falsa partilha pois a alteração de uma variável é possível sem afectar as outras [Tan95].

Nestes sistemas, o foco principal passa a estar relacionado com a manutenção de uma base de dados distribuída, potencialmente replicada. Ao contrário dos sistemas baseados em páginas, a utilização de um protocolo do tipo ‘write-update’ é mais interessante.

6.1 O Estudo do Caso 'Munin'

Este sistema, sendo baseado em variáveis e gerido por um sistema de suporte à execução, integra a MMU através da distribuição das variáveis partilhadas pelas páginas numa relação de 1:1 [Tan95].

O modelo de consistência implementado é o de consistência *'release'* (ver sec. correspondente), que resumidamente diz que enquanto um processo estiver activo dentro de uma região crítica, não existem garantias quanto à consistência; quando o processo sai da região crítica, as alterações são propagadas para todos os processadores.

Assim, o Munin distingue entre três tipos distintos de variáveis:

- ordinárias: apenas podem ser manipuladas pelo processo que as criou;
- partilhadas: podem ser utilizadas por vários processos, dando a impressão de estarem sequencialmente consistentes desde que utilizadas dentro de regiões críticas;
- sincronização: do tipo *'lock'* e *'barriers'*, apenas podem ser acedidas através de primitivas próprias.

Além disso, as variáveis partilhadas podem, por forma a aumentar o desempenho, ser classificadas explicitamente pelo programador aquando da sua declaração, de acordo com as seguintes categorias:

- *'read-only'*: não é possível alterar o seu valor, não levantando portanto problemas de consistência;
- *'migratory'*: variáveis não replicadas que migram entre os vários processadores, obrigam à aquisição de um *'lock'* que, por questões de desempenho, deve ser colocado na mesma página que a variável;
- *'write-shared'*: tipo de variáveis onde é admissível a alteração simultânea por parte de vários processadores, desde que a partes distintas da variável (por exemplo dois processadores a alterar duas sub-matrizes, distintas, de uma matriz).
- convencional: variáveis tratadas segundo um protocolo de replicação de leitura em tudo semelhante ao utilizado nos sistemas baseados em páginas.

Sempre que se trata de uma variável do tipo *'write-shared'*, quando os vários processos atingem um ponto de sincronização, é calculada a diferença entre as duas cópias alteradas e uma cópia original previamente guardada. É apenas esta diferença que é propagada para os restantes processadores.

Por forma a melhorar o desempenho do sistema, é possível, explicitamente por parte do programador, agrupar várias variáveis numa única página, desde que pertencentes ao mesmo tipo.

No que respeita à localização das várias cópias e respectivos donos (que no caso de variáveis com partilha de escrita não é necessariamente apenas um), o Munin utiliza um protocolo baseado em directorias, recorrendo à noção de *'dono provável'*, anteriormente introduzida.

No que respeita às variáveis de sincronização, os *'locks'* são implementados de uma forma distribuída sendo tratados como variáveis convencionais, ao passo que as *'barriers'* são implementadas por um servidor central. A localização das variáveis de sincronização é mantida numa directoria separada.

7. Sistemas Baseados em Objectos

O terceiro tipo de sistemas existente representa uma abordagem bastante estruturada de implementação de DSM. Assim, indo ao encontro do paradigma da programação mais em voga nos dias que correm, utiliza o conceito de objectos como unidade de partilha. Em contraste com

os sistemas de Memória Virtual Distribuída, implementados no 'kernel' com algum suporte por parte do hardware, os Sistemas Baseados em Objectos são implementados em software, fora do sistema operativo, recorrendo a um sistema de suporte à execução para a manutenção da consistência com base em informação gerada pelo compilador.

7.1 O Estudo do Caso 'Orca'

Este sistema [LKBT93] [Tan95] consiste numa linguagem, um compilador e um sistema de suporte à execução, com o objectivo de proporcionar um ambiente no qual processos em diferentes máquinas tenham um acesso controlado a uma DSM constituída por objectos.

Não fazendo parte do espectro deste trabalho, uma análise da linguagem e do compilador, a descrição do sistema resume-se ao sistema de suporte à execução ('runtime system'-RTS).

Assim, cabe ao RTS a manutenção do estado de consistência, obedecendo, neste caso, ao modelo sequencial, bem como a gestão da replicação, migração e invocação de operações. O facto do acesso aos objectos apenas poder ser feito através dos métodos definidos, aliado à atomicidade com que estes métodos são executados, tornam a sincronização do acesso bastante simplificada.

7.1.1 Gestão dos Objectos

Cada objecto pode existir num dos seguintes estados: cópia única ou replicado, podendo o seu estado variar durante a execução dum programa. As vantagens da replicação já foram largamente referidas. Assim, sempre que é efectuada uma operação sobre um objecto, é invocado o RTS, com indicação sobre se haverá alteração do objecto ou se apenas se trata de uma operação de leitura (esta informação é obtida pelo compilador). A forma como é executado o pedido depende também do estado do objecto, podendo ser distinguidas quatro situações distintas (ver fig 15):

- caso a única cópia do objecto resida localmente, é efectuado um 'lock', feita a operação e por fim o objecto é libertado (ver fig 15a);
- caso a única cópia resida noutra processador, o RTS efectua um RPC pedindo a execução da operação (ver fig 15b);
- caso o objecto esteja replicado e a operação apenas envolva uma leitura, ela é feita localmente sem necessidade de qualquer tráfego de rede(ver fig 15c);
- caso exista replicação do objecto e a operação envolva alteração do estado do mesmo, existem duas alternativas baseadas no facto de haver ou não um sistema de difusão eficiente:
 - no caso afirmativo, é efectuada uma difusão contendo o nome do objecto, a operação e os parâmetros, para que todos os processadores incluindo ele próprio efectuem a operação(ver fig 15d);
 - caso contrário, é enviada uma mensagem à cópia principal para que esta efectue um 'lock' de todas as cópias do objecto. Após a recepção das confirmações de todas as cópias, o processo original entra numa segunda fase em que envia outra mensagem pedindo a realização da alteração e desbloqueio dos objectos.

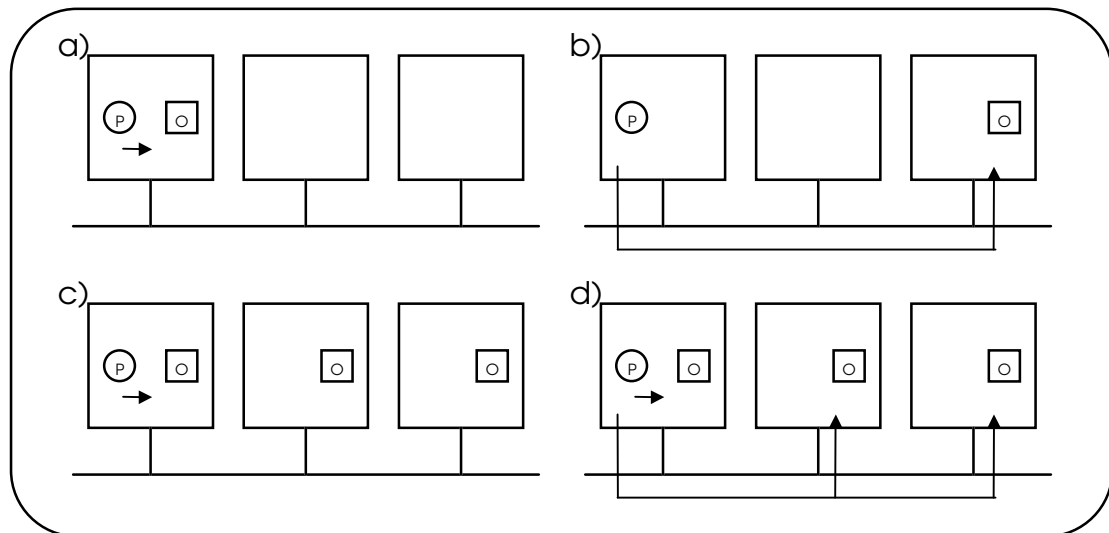


fig... operações sobre um objecto a)cópia única, local b)cópia única, remota c)replicado, leitura d)replicado, escrita (adaptado de [Tan95])

Desta forma, através dum algoritmo de ‘write-update’, consegue-se obter a implementação do modelo de consistência sequencial de uma forma bastante eficaz, evitando simultaneamente a existência de ‘deadlocks’ pelo facto de apenas um processo conseguir fazer o ‘lock’ da cópia principal.

Uma ultima referência para o facto da decisão sobre se um dado objecto deve ser replicado, ou não, estar dependente de uma análise ao programa, durante o processo de compilação.

7.2 Outros Sistemas

De entre os sistemas que implementam DSM baseada em ‘objectos’, será talvez importante destacar o sistema Linda [CG89] pelo conceito de objectos imutáveis que introduz. Assim, segundo este sistema, o espaço de memória partilhada é composto por um conjunto de ‘tuplos’ (estruturas semelhantes às estruturas do C) imutáveis. Ao programador são oferecidas quatro primitivas: *in*, *out*, *read* e *eval*, como sendo as únicas formas de aceder ao espaço partilhado, com as seguintes funções:

- *in*: retirar um dado ‘*tuplo*’ do espaço partilhado;
- *out*: introduzir um dado ‘*tuplo*’ no espaço partilhado;
- *read*: ler o valor de um ‘*tuplo*’, sem o retirar do espaço;
- *eval*: utilizado para computações em paralelo.

Assim, para por exemplo alterar um dado ‘*tuplo*’, ele tem de ser retirado do espaço partilhado, alterado e posteriormente reintroduzido.

8. Discussão

Ao longo deste trabalho foram abordadas várias formas de implementar um modelo de comunicação por memória partilhada sobre um sistema de troca de mensagens. Não sendo possível uma comparação quantitativa entre os vários sistemas, segue-se uma breve análise sobre a forma como as várias opções possíveis afectam alguns aspectos importantes na avaliação de um sistema de DSM.

Facilidade de Programação

A facilidade de programação de um sistema DSM está directamente relacionada com o modelo de consistência adoptado e com a estrutura utilizada para os dados partilhados.

Nos sistemas baseados em páginas, ao programador é exigido algum conhecimento sobre o funcionamento do sistema por forma a obter um bom desempenho, nomeadamente no que se refere à distribuição dos dados pelas várias páginas para evitar o fenómeno da falsa partilha. Por outro lado, a exclusão mútua é obtida explicitamente através das primitivas de sincronização.

Nos sistemas baseados em variáveis, o problema da falsa partilha é eliminado à custa de uma sobrecarga para o programador que é responsável pela determinação do tipo de variáveis sob a forma de anotações no programa.

Nos sistemas baseados em 'objectos' a exclusão mútua é fornecida implicitamente com as operações sobre os dados partilhados. Estes sistemas fornecem ainda alguma protecção a erros, nomeadamente através da detecção, por parte do sistema de suporte à execução, de utilização incorrecta de tipos de variáveis. A maior desvantagem de sistemas como por exemplo o 'Orca' reside no facto de utilizar uma linguagem de programação nova, ao contrário dos outros sistemas que podem ser programados recorrendo às linguagens existentes, com apenas algumas alterações.

No que se refere à influencia do modelo de consistência temos que, como já foi referido, quanto mais fraco for o modelo adoptado, mais pesada se torna a programação pelo facto de ser necessário a utilização explícita de mecanismos de sincronização para se obter a consistência sequencial.

Tolerância à Escala

Um aspecto fundamental sempre que se fala em sistemas com múltiplos processadores, sejam eles multiprocessadores ou multicomputadores, é a sua capacidade de funcionamento em larga escala. Sendo este um dos principais atractivos dos multicomputadores, é importante que o sistema de DSM consiga acompanhar essa característica e funcionar de uma forma eficiente em larga escala.

No que respeita à influencia do modelo adoptado, os centralizados, embora bastante eficientes para pequenos sistemas, tornam-se impraticáveis em grandes sistemas pelo congestionamento que provocam no gestor central. Quanto aos modelos distribuídos, embora resolvendo o problema anterior, necessitam de um maior número de mensagens por operação; sabendo que quanto menor for a largura de banda necessária melhor a tolerância à escala, este pode ser um factor negativo.

Outro factores que influenciam a tolerância à escala são:

- o padrão de acesso de cada aplicação na medida em que, quanto maior for a relação escritas/leituras menor será a capacidade do sistema crescer porque as leituras podem ser feitas localmente, nas réplicas, sem a necessidade de utilização de mensagens;
- a granularidade, ou seja, num sistema em que as operações sobre a DSM sejam de baixo nível, como nos sistemas baseados em páginas, existirá uma grande quantidade de tráfego de rede. Num sistema mais estruturado, como sejam os baseados em 'objectos', as operações são de alto nível (métodos, procedimentos) reduzindo o número de mensagens necessárias;
- o modelo de consistência sendo que, quanto mais fraco for o modelo adoptado menor o número de mensagens necessárias e consequentemente melhor será a tolerância à escala do sistema.

Desempenho

É extremamente difícil fazer uma avaliação comparativa entre as várias soluções apresentadas no que respeita ao desempenho por ser um factor que está decisivamente dependente da

aplicação em causa. Temos por exemplo que se uma variável depois de ser alterada, for requerida por um grande número de outros processadores, a política recomendada seria a de 'write-update'; se ao contrário, essa variável não for requerida por um elevado número de processadores, então uma política de 'write-invalidate' seria mais eficaz.

O desempenho é também afectado pelo modelo de consistência adoptado; assim, quanto mais fraco for o modelo de consistência, melhor o desempenho.

Tolerância a falhas

Este aspecto, embora bastante importante, não foi abordado por nenhum dos sistemas apresentados. É no entanto necessária alguma forma de lidar com situações como por exemplo a de um processador que contém uma réplica falhar e conseqüentemente não responder ao pedido de invalidação/actualização. Algumas soluções possíveis passam pelo tratamento do sistema como uma base de dados distribuída, área onde existe bastante trabalho feito em termos de investigação.

9. Conclusões

Sendo uma área de investigação relativamente recente, os progressos obtidos são bastante encorajadores no que respeita à possibilidade de implementar, em larga escala e de uma forma eficiente, o paradigma da comunicação por memória partilhada num sistema multicomputador. No entanto, devido às dificuldades actuais de realização eficiente deste modelo de memória, o modelo de troca de mensagens continua a ser o dominante em sistemas com memória distribuída.

A investigação futura nesta área estará certamente ligada à questão da tolerância à escala, permitindo tirar partido da grande capacidade computacional das grandes redes de computadores, bem como à questão da heterogeneidade por forma a permitir a interligação de sistemas diferentes.

10. Bibliografia

- [BZS93] Bershada, B., Zekauskas, M., Sawdon, W. (1993). *The Midway Distributed Shared Memory System*. COMPCON, 1993
- [CG89] Carriero, N. and Gelernter, D. (1989). *Linda in Context*. Communications of the ACM, Vol. 32 No. 4, April 1989
- [CGB91] Cheriton, D., Goosen, H. and Boyle, P. (1991). *ParaDiGM: A Highly Scalable Shared-Memory Multicomputer Architecture*. IEEE Computer, pp 33-46, February 1991
- [CDK94] Coulouris, G., Dollimore, J. and Kindberg, T. (1994). *Distributed Shared Memory*. In *Distributed Systems - Concepts and Design*, 2nd Edition. Addison-Wesley. pp. 517-544
- [KS95] Kontothanassis, L. and Scott, M. (1995). *Distributed Shared Memory for New Generation Networks*. Technical Report 578, University of Rochester
- [LM92] LeBlanc, T. and Markatos, E. (1992). *Shared Memory vs. Message Passing in Shared-Memory Multiprocessors*. Technical Report, University of Rochester
- [LKBT93] Levelt, W., Kaashoek, M., Bal, H. and Tanenbaum, A. (1993). *A Comparison of Two Paradigms for Distributed Shared Memory*. Technical Report, Dep. C. S. Vrije Universiteit
- [LH89] Li, K. and Hudak, P. (1989). *Memory Coherence in Shared Virtual Memory Systems*. ACM Trans. on Computer Systems, Vol. 7, No. 4
- [MF89] Minnich, R., Farber, D. (1989). *The Mether System: Distributed Shared Memory for SunOS 4.0*. Proc. Summer 1989 Usenix Conf.
- [Moh93] Mohindra, A. (1993). *Issues in the Design of Distributed Shared Memory Systems*. Ph.D. Thesis, Georgia Institute of Technology, May 1993

- [Mos93] Mosberger, D. (1993). *Memory Consistency Models*. Technical Report 93/11 University of Arizona
- [RS93] Ramachandran, M. and Singhal, M. (1993). *On the Synchronization Mechanisms in Distributed Shared Memory Systems*. Technical Report, Ohio State University
- [SS92] Saulsbury, A. and Stiernerling, T. (1992). *A DVSM Server for MESHIX*. Technical Report, City University
- [SMS89] Shaffer, J., Minnich, R. and Smith, J. (1989). *Architecture and Performance of the Mether Network Shared Memory*.
- [Sin93] Sinha, H. (1993). *Mermera: Non-Coherent Distributed Shared Memory for Parallel Computing*. Ph.D. Thesis, Boston University, 1993
- [SZ90] Stumm, M. and Zhou, S. (1990). *Algorithms Implementing Distributed Shared Memory*. IEEE Computer, pp 54-64, May 1990
- [TSF90] Tam, M., Smith, J. and Farber, D. (1990). *A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems*. ACM Operating Systems Review, Vol. 24, pp.40-67, July 1990
- [Tan95] Tanenbaum, A. S. (1995). *Distributed Shared Memory*. In Distributed Operating Systems. International Edition. Prentice Hall. pp. 289-375
- [WLT93] Wilson Jr., A., LaRowe Jr., R. and Teller, M. (1993). *Hardware Assist for Distributed Shared Memory*. IEEE Proc. 13th International Conference on Distributed Computing Systems, May 1993