Desempenho Simulado de Modelos Fracos de Consistência de Memória

M. C. R. Carneiro [1] R. C. G. Pinto [2] C. B. Seidel [3] A. B. R. da Silva [4]
M. G. da Silva [5] C. L. de Amorim [6]

COPPE/Sistemas
Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brasil

RESUMO

Os dois modelos de consistência: *Release* e *Entry*, representam duas soluções distintas para o problema de coerência em sistemas de memória compartilhada distribuída. *Release Consistency* oferece um modelo viável para implementação em hardware enquanto que *Entry Consistency* é mais flexível e orientado para software. Neste trabalho simuladores dos dois modelos são desenvolvidos e testados numa rede de workstations utilizando PVM. Os modelos *Release* e *Entry* são avaliados baseando-se nas implementações DASH e Midway, respectivamente, utilizando-se uma rede com 2, 4 e 8 nós. Os resultados preliminares são contrastados com o desempenho de um sistema de memória distribuída equivalente e indicam que versões híbridas, ou seja, implementações preguiçosas em software de modelos de consistência de memória com suporte de hardware podem oferecer soluções atrativas.

ABSTRACT

Release Consistency and Entry Consistency represent two distinct solutions to the memory coherence problem in distributed shared memory systems. Release Consistency offers a viable model for hardware implementation whereas the Entry Consistency is more flexible and software oriented. In this work simulators for the two models are developed and tested using a network of workstations and PVM. By using the DASH and the Midway implementations we evaluate Release and Entry models, respectively, in a network with 2, 4 and 8 nodes. The preliminary results are contrasted with the performance of an equivalent distributed shared memory and they indicate that hybrid versions, that is, software implementations of lazy consistency memory models with hardware support can offer attractive solutions.

[1] Aluna de mestrado COPPE/Sistemas; email:carol@cos.ufrj.br

[2] MSc (COPPE-1994); Pesquisadora COPPE-Sistemas; email:raquel@cos.ufrj.br

[3] MSc (COPPE-1991),Aluna de doutorado COPPE/Sistemas; Professora Assistente da Universidade do Estado do Rio de Janeiro; email:seidel@cos.ufrj.br

[4] Aluno de mestrado COPPE/Sistemas; email:bellieny@cos.ufrj.br

[5] Aluno de mestrado COPPE/Sistemas; email: mgs@cos.ufrj.br

[6] PhD (Imperial College - 1984), MSc (COPPE - 1979); Professor Adjunto da COPPE/Sistemas; email: amorim@cos.ufrj.br

* Esse trabalho foi parcialmente financiado pela Finep, CNPq e Capes

1 Introdução

O modelo mais simples de programação paralela é aquele que assume a existência de um espaço de endereçamento global para todos os processos. Esse modelo de programação é típico dos multiprocessadores com memória compartilhada, tais como o Onyx da Silicon Graphics e o Balance da Sequent. Entretanto, arquiteturas desse tipo não permitem o emprego de um grande número de processadores.

A implementação do modelo de memória compartilhada em sistemas com memória distribuída tem se mostrado uma alternativa bastante promissora. Sistemas desse tipo possuem mecanismos de hardware e/ou software que transformam automaticamente os acessos à "memória compartilhada" em acessos às memórias locais ou às memórias remotas. São chamados de **sistemas com memória compartilhada distribuída** e oferecem um modelo de programação relativamente simples em um hardware escalável.

O uso de memórias privativas, porém, requer solução para o **problema de consistência de memórias**. Várias cópias de um mesmo dado podem residir em memórias locais de diferentes processadores simultaneamente, e se for permitido aos processadores atualizar livremente suas cópias, uma visão inconsistente da memória poderá ser formada, levando o programa a resultados imprevisíveis.

Uma grande atenção tem sido dada nos últimos anos para o problema de consistência de memórias porque sua solução tem grande impacto no desempenho do sistema como um todo. Diversos esquemas foram propostos para resolver esse problema. Alguns são implementados em software e se baseiam em ações tomadas pelo programador, pelo compilador ou pelo sistema operacional e outros são fortemente suportados pelo hardware da máquina. Soluções de hardware são mais transparentes para o programador ou compilador, enquanto que soluções de software são mais baratas e adaptáveis.

Para garantir a consistência das memórias é necessário incluir no sistema certas restrições na forma como os processadores acessam os dados compartilhados. Os modelos de consistência existentes apresentam diferentes tipos de restrições. Estudos [8, 15] mostram que modelos mais fracos de consistência (i.e, que têm menos restrições) apresentam melhor desempenho porque conseguem reduzir boa parte da latência dos acessos de escrita a dados compartilhados.

Neste trabalho avaliamos o comportamento de dois modelos fracos de consistência de memória, o modelo *Release* e o modelo *Entry*, num sistema com memória compartilhada distribuída. A fim de estudar as características de implementações em hardware e em software do modelo de memória compartilhada distribuída, utilizamos implementação da consistência em hardware para o modelo *Release*, tal qual o sistema DASH descrito em [13], e implementação em software para o modelo *Entry*, tal qual o sistema Midway descrito em [3].

As recentes pesquisas em sistemas de memória compartilhada distribuída mostram as vantagens de utilizar programas escritos para memória compartilhada (que assumem um espaço de endereçamento global) em máquinas com memória distribuída. Nossos experimentos, entretanto, apresentam um outro enfoque. Estamos interessados em comparar a eficiência de duas versões, compartilhada e distribuída, de uma mesma aplicação. Na versão distribuída (baseada em troca de mensagens), a gerência de toda a comunicação é feita pelo programador, conse-

quentemente os programas apresentam uma quantidade mínima de troca de mensagens. Na versão compartilhada, a troca de mensagens é gerada pelos protocolos utilizados para garantir a consistência de memória. Comparando-se as duas versões de uma mesma aplicação, estaremos investigando o *overhead* gerado pelos protocolos de consistência de memória e a diferença dos dois modelos de programação.

Na próxima seção apresentamos as definições dos modelos de consistência usados no trabalho. Em seguida, na seção 2, descrevemos brevemente os simuladores de cada modelo, mostrando como são implementadas operações de acessos a dados compartilhados e operações de sincronização. Na seção 4 estão descritos os resultados da simulação, utilizamos duas aplicações e testamos as suas versões compartilhada e distribuída. Finalmente na seção 5 apresentamos as conclusões do trabalho.

2 Modelos de Consistência

A solução do problema de consistência de memória envolve a definição de um modelo que especifique formalmente como os acessos à memória compartilhada realizados por um processador são observados pelos outros processadores do sistema. O modelo mais simples e intuitivo de consistência é o chamado **consistência seqüencial**. Ele foi formalizado por Lamport [12]: " Um sistema é seqüencialmente consistente se o resultado de qualquer execução é o mesmo que se as operações de todos os processadores fossem executadas em alguma ordem seqüencial, e as operações de cada processador na ordem descrita pelo programa". Em outras palavras, em termos de acessos a dados compartilhados, um multiprocessador sequencialmente consistente funciona como se fosse um único processador multiprogramado.

O modelo de consistência seqüencial, entretanto, restringe muito o paralelismo potencial do sistema. Um acesso a um dado compartilhado só pode ser realizado se o acesso anterior tiver sido observado por todos os outros processadores, o que deixa o processador paralisado por um período de tempo. Além disso, este modelo apresenta alguns obstáculos à implantação de certos mecanismos de hardware usados para fins de otimização, como por exemplo, *pipelines* de escrita ou mecanismos de *prefetch*.

Garantir a consistência da memória empregando um modelo menos restritivo pode, portanto, ter grande influência no desempenho do sistema.

É possível relaxar as restrições impostas pelo modelo seqüencial quando o programador deixa explícito no seu programa os pontos em que ele acha que a consistência dos dados é necessária, isto é feito através do uso de variáveis de sincronização. Nesses programas podem ser empregados modelos **fracos** de consistência de memória. Diversos modelos fracos de consistência de memória encontram-se na literatura [6, 7, 1, 9, 3, 2]. Os modelos *Release* e *Entry* foram escolhidos para nossas implementações porque são os mais fracos e possuem implementações eficientes em hardware (para o *Release*) e em software (para o *Entry*) descritas na literatura.

2.1 O Modelo *Release*

O modelo *Release* foi desenvolvido por Gharachorloo *et al* [9]. Ele garante que a memória é consistente apenas em determinados pontos de sincronização. Os pontos de sincronização referem-se a acessos a variáveis de sincronização, sendo que esses

acessos são classificados como *acquires* e *releases*. Um *acquire* indica que o processador está iniciando uma operação que pode depender de valores gerados por outro processador, por exemplo, uma operação *lock* na entrada de numa seção crítica. A execução de um *release* indica que o processador está terminando uma operação que gerou valores dos quais outro processador pode depender, por exemplo, uma operação *unlock* na saída de uma seção crítica. De uma forma geral, *acquires* representam acessos de leitura a uma variável de sincronização e *releases* representam acessos de escrita a uma variável de sincronização.

As condições formais para que um sistema esteja consistente segundo o modelo *Release* são:

- Antes de executar um *release*, todos os acessos a dados compartilhados anteriores devem ter sido observados por todos os processadores;
- Acessos que seguem uma operação de *acquire* numa variável de sincronização s devem esperar que o *acquire* tenha terminado. Um *acquire* a s está terminado quando escritas posteriores em s realizadas por outros processadores não afetam o valor lido no *acquire*.
- Acessos a variáveis de sincronização são seqüencialmente consistentes;

Segundo essas condições, no momento da execução de um *release* é necessário garantir que as escritas anteriores já foram observadas pelos outros processadores. Ou seja, a saída de uma seção crítica serve para sinalizar outros processadores que a seção crítica está livre e que tudo o que foi feito dentro dela deve ser visto por todos os processadores. E ainda, quando um processador vai executar um *acquire* e entrar em uma seção crítica, as escritas realizadas pelos outros processadores devem estar visíveis localmente.

Em relação ao modelo de consistência sequencial, o modelo *Release* apresenta uma sensível redução na latência de acesso à memória compartilhada por dois motivos principais. Primeiro devido a diminuição do tempo que um processador fica paralisado por causa da consistência. Os pontos de espera são apenas os *releases*. *Acquires* e acessos a dados compartilhados não causam espera por motivos de consistência (os *acquires* apenas atrasam acessos futuros). Segundo porque como as leituras e escritas realizadas entre acessos a variáveis de sincronização podem ser observadas em qualquer ordem por outros processadores, as escritas podem ser feitas em *pipeline* e as leituras podem passar as escritas (desde que obedeçam as dependências dentro do programa) permitindo a utilização de um mecanismo de *prefetch*.

O modelo *Release* de consistência foi implementado em diversos sistemas. No projeto DASH [13] da Universidade de Stanford, é o hardware que garante a consistência das memórias e o modelo *Release* é implementado usando um esquema baseado em diretórios. O sistema Munin desenvolvido por Carter *et al* [5] é um sistema de programação para memória compartilhada distribuída que provê a consistência por software. Os protocolos usados no sistema Munin diferem dos usados no DASH pela seguinte característica: as escritas realizadas dentro de uma seção crítica são atrasadas até o momento da execução do *release*, i.e, elas só são enviadas para os outros processadores imediatamente antes da execução do *release*, o que

diminui o número de mensagens enviadas (já que as atualizações vão todas em uma única mensagem).

O trabalho de Keleher *et al* [10] apresenta uma versão "preguiçosa" do modelo *Release* que eles chamaram de *Lazy Release*. Neste modelo o protocolo utilizado para manter a consistência atrasa ao máximo o envio de escritas para outros processadores a fim de diminuir o número de mensagens e a quantidade de dados trocados. As escritas realizadas antes de um *release* no processador p_1 não são enviadas para os outros processadores, mas quando um processador p_2 vai executar um *acquire* na mesma variável de sincronização, ele deve receber numa mensagem as escritas realizadas em p_1. Assim, as escritas são enviadas somente quando necessárias para a realização de um *acquire*. O trabalho de Kontothanasssis *et al* [11] mostra a viabilidade de se implementar um modelo "preguiçoso" em hardware.

2.2 O Modelo *Entry*

O modelo *Entry* de consistência de memória foi desenvolvido por Bershad *et al* [3, 4]. Nesse modelo, a cada dado compartilhado é associada uma variável de sincronização. A variável de sincronização que controla o acesso à seção crítica é chamada de **guarda** do dado compartilhado. Um dado compartilhado estará consistente quando for realizado um *acquire* na variável de sincronização que o guarda.

O modelo *Entry* diferencia também os acessos a variáveis de sincronização como sendo de modo **exclusivo** ou de modo **não-exclusivo**. Um acesso a uma variável de sincronização no modo não-exclusivo permite que vários processadores estejam simultaneamente executando a mesma seção crítica, o que deve ser usado quando são realizadas somente leituras no dado. A atualização de um dado compartilhado deve ser feita no modo exclusivo.

Sendo s uma variável de sincronização que guarda o dado D_s, a condição formal para que um sistema esteja consistente segundo o modelo *Entry* é:

- Um acesso *acquire* a s só pode ser observado em p_i quando todas as atualizações a D_s foram observadas em p_i;

O modelo *Release* descrito anteriormente requer que antes da execução de um *release* as atualizações feitas em qualquer dado compartilhado sejam observadas pelos outros processadores, o que inclui dados não guardados pela variável de sincronização que está sendo liberada. E como essas atualizações devem ser enviadas para os ou tros processadores antes da liberação da seção crítica, o tempo de execução da seção crítica aumenta. Já o modelo *Entry* envia para os outros processadores apenas as atualizações dos dados guardados pela variável de sincronização a ser liberada e este envio só precisa ser realizado no próximo *acquire* na mesma variável (como na versão "preguiçosa" do *Release*). Assim, são transferidos apenas os dados necessários e esta transferência ocorre toda fora da seção crítica.

A utilização de *acquires* exclusivos e não-exclusivos tem como vantagem o fato que só é necessária comunicação entre dois acessos *acquires* exclusivos. Não há necessidade de troca de mensagens quando um mesmo processador realiza vários *acquires* não-exclusivos no mesmo dado.

O sistema Midway [3] é um sistema de programação para memória compartilhada distribuída que implementa o modelo *Entry* de consistência de memória. O protocolo

utilizado é bem simples, antes da execução de um *acquire* em um guarda s no processador p_1, este deve receber todas as atualizações realizadas em D_s.

Apesar do modelo *Entry* gerar uma quantidade bem menor de trocas de mensagens ele requer um modelo de programação mais complicado devido à necessidade de se estabelecer um guarda para cada dado compartilhado e de se diferenciar acessos em modo exclusivo e não-exclusivo.

3 Implementação dos Modelos de Consistência

Esta seção descreve como foram simulados os modelos *Release* e *Entry* num sistema distribuído composto por um conjunto de estações de trabalho Sun SPARC que utilizam as primitivas PVM (*Parallel Virtual Machine*) para trocas de mensagens. Os simuladores foram escritos na linguagem C e provêem ao programador um conjunto de funções que dão suporte ao modelo de programação paralela com memória compartilhada. Estas funções incluem a realização de acessos de escrita e leitura a dados compartilhados e a implementação de primitivas de sincronização do tipo: *lock/unlock* e barreira.

3.1 Simulador do Modelo *Release*

Nossa implementação do modelo *Release* de consistência de memória segue o protocolo utilizado no sistema DASH [13]. Nesse sistema a consistência de memória é garantida pelo hardware da máquina.

A arquitetura simulada é composta de um conjunto de processadores ligados por uma rede de interconexão. A memória é fisicamente distribuída pelos processadores, mas acessível por todos eles e cada processador possui uma memória cache privativa.

O simulador do modelo *Release* utiliza um conjunto de protocolos baseados em trocas de mensagens para garantir que no momento da execução de um *release*, todas as atualizações realizadas dentro de uma seção crítica foram observadas por todos os outros processadores. Dessa forma, o efeito da latência das escritas pode ser atrasado até o momento da execução de um *release*.

É utilizado um esquema de **invalidação** para que uma escrita em s no processador p_i seja observada em p_j. Isto é, p_i envia para p_j uma mensagem invalidando s e somente no próximo acesso a s em p_j, o valor de s é efetivamente buscado. Os acessos de escrita realizados dentro de uma seção crítica propagam invalidações para os outros processadores e a execução de um *release* paralisa o processador até que todas as invalidações enviadas tenham sido recebidas.

Com relação às funções de programação paralela oferecidas pelo simulador, estão disponíveis primitivas de leitura (*rc-read*) e escrita (*rc-write*) para dados compartilhados e para sincronização entre processos estão disponíveis primitivas de *lock/unlock* (*rec-lock/rc-unlock*) e de barreira (*rc-barrier*).

A função *rc-init* inicia a operação de um simulador em cada processador e determina um identificador único para cada simulador iniciado. É por esta função que o programador determina o tamanho da memória compartilhada e o número de processadores a serem utilizados na aplicação.

A função *rc-fork* é responsável por disparar os processos paralelos em cada processador. Cada processo disparado é associado ao simulador que executa no mesmo

processador, recebendo o mesmo identificador. Este identificador pode ser capturado através da função *rc-myid*.

O simulador trabalha com uma distribuição estática de dados. A memória compartilhada é dividida entre os processadores de maneira uniforme, sendo cada processador o responsável pelas informações de cada endereço dentro do seu trecho de memória compartilhada. Diz-se então que este processador é o **supervisor** destes endereços.

Todas as transações envolvendo dados devem ser feitas através de blocos de memória. O simulador utiliza um tamanho de bloco de 32 bytes. Cada supervisor deverá ter, então, uma estrutura de dados contendo as informações de cada bloco nos outros processadores, a que chamamos de **diretório**. Cada entrada de diretório indica quais processadores têm cópia deste bloco e o estado da cópia.

Um bloco pode estar em três estados em relação aos processadores que não são o supervisor deste bloco: *sem-cópia-remota*, *copiado-remotamente*, *modificado-remotamente*. *Sem-cópia-remota* significa que não há cópia deste bloco em nenhum outro processador. *Copiado-remotamente* significa que pelo menos um processador tem cópia de leitura deste bloco. E *modificado-remotamente* indica que um processador tem cópia deste bloco para leitura e escrita. O simulador permite leituras concorrentes mas quando um processador escreve, apenas ele pode ter cópia válida deste bloco. Diz-se então que tal processador é o **proprietário** deste bloco.

Um bloco pode estar em três estados dentro de um determinado processador: *invalido*, *copiado* ou *modificado*. *Invalido* significa que não é possível fazer qualquer acesso a um endereço neste bloco. *Copiado* indica que é possível fazer um acesso de leitura a um endereço dentro deste bloco. *Modificado* indica que este processador pode fazer um acesso de leitura ou escrita em qualquer byte deste bloco.

Quando um processador necessita acessar a memória compartilhada e sua cópia não é válida para aquele acesso (por exemplo, o processador tem cópia de leitura de um bloco mas quer fazer um acesso de escrita, ou não tem cópia válida e quer fazer um acesso qualquer), ele deve requisitar este bloco ao processador supervisor. Diz-se então que o processador que necessita fazer o acesso é o requerente deste bloco.

Implementação da Operação de Leitura

Supondo que um processador p_i faz um acesso de leitura a um dado D, esse acesso vai gerar um conjunto de trocas de mensagens segundo os casos descritos abaixo:

- Quando o p_i não possui cópia de leitura para o bloco que contém D, ele envia uma mensagem RDREQUEST para o supervisor.
- O supervisor verifica as informações sobre o bloco que contém D no seu diretório:
  - Se o bloco estiver *sem-cópia-remota* ou *copiado-remotamente* ele envia para p_i o bloco e atualiza seu diretório.
  - Se o bloco estiver *modificado-remotamente* ele avança o pedido de leitura para o processador proprietário de D. O proprietário, então, envia resposta para p_i contendo o bloco e envia também uma mensagem de SHWRI-

TEBACK com o bloco para o supervisor para que ele atualize seu diretório indicando que p_i e o antigo proprietário de D têm cópia de leitura e ao mesmo tempo atualizando a cópia do supervisor.

Implementação da Operação de Escrita

Supondo que um processador p_i faz um pedido de escrita em D, esse pedido vai gerar um conjunto de trocas de mensagens segundo os casos descritos abaixo:

- Quando p_i não possui cópia de escrita para o bloco que contém D, ele aloca uma entrada na Tabela de Invalidações Pendentes (TIP) indicando que o processador possui um número de invalidações a serem recebidas, inclui a escrita pendente numa lista de escritas, envia uma mensagem RDEXREQUEST para o supervisor e segue o processamento normal (o que possibilita realizar a atualização em paralelo com processamento, reduzindo a latência relativa ao acesso de escrita).
- O supervisor verifica as informações sobre o bloco que contém D no seu diretório:
  - Se o bloco estiver *sem-cópia-remota*, ele envia o bloco para p_i com número de invalidações igual a 0 e atualiza seu diretório indicando que o bloco está *modificado-remotamente*.
  - Se o bloco está *copiado-remotamente*, ele envia o bloco para p_i com o número de invalidações a serem recebidas por p_i. Além disso, ele envia para cada processador que possui cópia do bloco no estado *copiado*, uma mensagem de INVREQUEST. Ao receber uma mensagem INVREQUEST o processador deve invalidar a sua cópia do bloco e enviar uma mensagem de INVACK para p_i.
  - Se o bloco está *modificado-remotamente*, ele avança o pedido de escrita para o processador proprietário de D. O processador proprietário envia o bloco para p_i com o número de invalidações igual a 1, envia uma mensagem DTTRANSFER com o bloco para o supervisor e invalida seu próprio bloco. Ao receber DTTRANSFER o supervisor atualiza seu diretório registrando a mudança de propriedade do bloco e envia uma mensagem INVACK para p_i, indicando que a transação já foi devidamente registrada. As entradas da TIP persistem até que todos os INVACK's tenham chegado, o que significa que todos estão cientes que p_i é o novo proprietário de D.

3.2 Simulador do Modelo *Entry*

O modelo *Entry* de consistência de memória foi implementado conforme o sistema Midway descrito em [3, 4]. O sistema Midway implementa uma memória compartilhada virtual num sistema distribuído.

O modelo *Entry* de consistência de memória apresenta restrições mais fracas que o modelo *Release*. O simulador do modelo *Entry* propaga as atualizações realizadas

em um determinado dado somente na execução do próximo *acquire* no mesmo dado. Os protocolos utilizados no simulador também utilizam esquema de **invalidação** na propagação das atualizações.

O simulador do modelo *Entry* é dividido em duas partes. A primeira delas suporta as funções utilizadas no programa que dispara os processos paralelos nos demais processadores. Chamamos essa primeira parte de simulador-pai. A segunda parte suporta as funções relativas à sincronização. Essa é simplesmente chamada de simulador. Os simuladores interagem entre si através da troca de mensagens.

Quando um processo é disparado para a execução da aplicação, ele dispara também o simulador-pai no mesmo processador. Entre as principais funções do simulador-pai usadas pela aplicação estão a associação de um dado a uma variável de sincronização e o disparo dos demais processos da aplicação. A função *ec-fork* dispara os processos que vão executar nos outros processadores. O numero de processos disparados, bem como o processador onde eles devem ser executados são parâmetros definidos pelo usuário. A função *ec-fork* dispara também um simulador em cada processador.

Para a implementação do modelo *Entry* de consistência de memória, é necessário que seja feita a associação de variáveis de sincronização (guardas) com os dados compartilhados. Existe no simulador-pai uma função própria para realizar esta associação que é chamada de (*ec-bind*). É no momento dessa associação que os dados passam a ser vistos pelo sistema como compartilhados. Variáveis de sincronização são representadas por números inteiros e declaradas somente no momento da inicialização dos dados compartilhados.

O simulador fornece ao usuário dois tipos de operações de sincronização: *lock/unlock* e as barreiras, sendo que *locks* podem ser adquiridos em modo exclusivo (pela função *ec-lock-ex*), para leitura e escrita, e em modo não-exclusivo (pela função *ec-lock-nex*), somente para leitura. A função *ec-release* implementa a operação de *unlock* e a função *ec-barrier* faz a realização de uma barreira.

Implementação das Operações *Lock/ Unlock*

Segundo o modelo de consistência *Entry*, o *lock* tem a mesma função de um *acquire* e, portanto, sua implementação envolve um conjunto de trocas de mensagens, porque um processo que está entrando numa seção crítica deve receber todas as modificações feitas ao dado compartilhado, guardado pela variável de sincronização sobre a qual está sendo feito o pedido de *lock*. O simulador deve, então, procurar entre os processadores, qual deles contém a cópia dos dados mais recentemente modificada e a transfere para o processador que fez o pedido.

Os locks exclusivos e não-exclusivos, foram implementados usando-se o conceito de *proprietário* de s. Dado que s é a variável de sincronização, que guarda o dado D_s, proprietário é o identificador do processador que possui, ou que possuiu por último, o acesso de *lock* exclusivo a s, e que, portanto, está modificando ou modificou D_s. E chamaremos de requerente o processador que está fazendo o pedido de *lock*.

O pedido de *lock* exclusivo em uma variável s é realizado da seguinte forma:

1. Quando o processador requerente é o proprietário de s:
   - Caso não hajam cópias de D_s, fornecidas para leitura, distribuídas pelos

processadores, o *lock* é adquirido imediatamente.

- No caso de existirem cópias de D_s, essas são invalidadas antes do *lock* ser adquirido, já que não é permitido aos processadores ler e modificar ao mesmo tempo. A invalidação das cópias é feita através de mensagens INVALID enviadas a todos os processadores que possuem o *lock* não exclusivo de s, e que, portanto, estavam fazendo leituras ao dado D_s. Assim que os processadores completam a invalidação, eles enviam ao proprietário uma mensagem RESP_INV de resposta. É o proprietário de s que controla quantos e quais os processadores que contém cópia de D_s. Quando chegarem ao proprietário todas as respostas, o processador requerente já pode adquirir o *lock* exclusivo de s e modificar D_s.

2. Quando o processador requerente não é o proprietário de s:

   - É enviada uma mensagem REQUEST ao proprietário de s. O proprietário então, vai tomar o mesmo procedimento de invalidação das cópias de D_s, descrito acima, caso hajam cópias de leitura. Depois, ele envia ao processador requerente uma mensagem RESP_REQ contendo o dado D_s em sua versão mais recentemente modificada, transformando o processador requerente no novo proprietário de s.

O pedido de *lock* não-exclusivo em uma variável s é realizado da seguinte forma:

- Se o processador requerente é proprietário de s, o *lock* é adquirido, lembrando que, mesmo sendo proprietário, o processo só poderá fazer leituras a D_s.

- Se não for proprietário, o processador requerente envia uma mensagem REQUESTN ao processador "mais perto" dele e que possua uma cópia de D_s, retornado pela mensagem RESP_REQN. No *lock* não exclusivo, não é necessário que essa cópia seja fornecida pelo proprietário, basta que seja um processador que possua uma cópia de leitura de D_s válida.

Assim que leituras ou modificações ao dado D_s terminam, o processo que adquiriu o *lock* já pode fazer o *unlock* (i.e., o *release*) da variável s, para que ela fique disponível aos demais processadores. Durante o tratamento de um *lock* exclusivo a s, os outros pedidos de *lock* exclusivo e não-exclusivo a s que chegam ao proprietário são enfileirados. Durante o tratamento de um *lock* não-exclusivo a s, os pedidos de *lock* exclusivo em s também são enfileirados, para que possam ser atendidos quando da liberação da variável s (no *unlock*).

Implementação da Barreira

Barreiras são utilizadas para sincronizar um conjunto de processadores. Quando um processador alcança uma barreira, ele permanece paralisado até que os demais processadores envolvidos na operação alcancem também a barreira. Os dados compartilhados que estão associados a uma barreira somente serão considerados consistentes no momento em que todos os processos atravessam a barreira. Isso porque a todos os processos da barreira é permitido fazer modificações no dado associado à barreira.

O conceito de **gerente** foi usado na implementação das barreiras. O gerente de uma dada barreira b é o identificador do processador que controla a variável b, o dado compartilhado D_b (associado a essa barreira) e quantos processos estão envolvidos nesta operação, que formam o conjunto P. A barreira é, então, implementada da seguinte forma:

- Um processo do conjunto P, assim que alcança a barreira b, envia uma mensagem ao gerente, indicando que alcançou a barreira. Junto com essa mensagem estão todas as modificações feitas pelo processo, ao dado D_b.
- O gerente, a medida em que vai recebendo essas mensagens, vai unindo todas as modificações feitas a D_b.
- Assim que as mensagens de todos os processos do conjunto P são recebidas, o gerente libera esses processos para darem prosseguimento ao seu processamento, através do envio de uma mensagem contendo todas as modificações de D_b recebidas e unidas.
- Os processos que fazem parte de P recebem a mensagem e integram as modificações à sua memória local.

4 Resultados

A seguir apresentamos os resultados obtidos pela execução de duas aplicações diferentes (multiplicação de matrizes e ordenação de listas) nos nossos simuladores. Escolhemos duas aplicações simples para a validação dos simuladores e para que fosse possível comparar as duas versões, compartilhada e distribuída, de cada aplicação. O uso de *benchmarks* padrões para memória compartilhada (como o conjunto SPLASH [14]) não foi considerado, num primeiro momento, porque essas aplicações não possuem versão distribuída conhecida.

4.1 Aplicações

Idealmente, a versão distribuída de uma dada aplicação deve apresentar um número mínimo de mensagens e uma quantidade mínima de bytes trocados. Isto porque é o programador o responsável por gerar toda a comunicação entre processos. Na versão compartilhada, entretanto, as mensagens são geradas pelos protocolos de consistência de memória.

Estamos interessados em investigar se o uso do modelo de programação com memória compartilhada inclui num sistema distribuído um *overhead* de comunicação muito alto em relação à versão distribuída da mesma aplicação.

Multiplicação de Matrizes

O algoritmo executa a multiplicação de duas matrizes quadradas A e B de dimensão m e guarda o resultado na matriz C. Estamos considerando que existem n processadores na máquina.

Na **versão compartilhada**, as matrizes A e C, serão acessadas por todos os processadores, para leitura e escrita dos dados. A matriz B é acessada pelos processadores da seguinte forma: cada processador é o proprietário inicial de um conjunto de m/n colunas diferentes de B e realiza somente leituras sobre essas colunas. A distribuição inicial das colunas da matriz B foi feita de forma a otimizar a execução da aplicação, evitando tráfego inútil de dados pela rede. Cada processador calcula os elementos de m/n colunas da matriz C (m deve ser divisível por n). A matriz C resultante se torna consistente e disponível para novos acessos depois que todos os processadores atravessam uma barreira.

Na **versão distribuída** da multiplicação de matrizes, inicialmente é realizada a distribuição dos dados enviando a matriz A para os n processadores e dividindo a matriz B em colunas de forma a enviar m/n colunas para cada processador. Os processos ao receberem estes dados, calculam suas respectivas colunas da matriz C e as enviam para o processador que realizou a distribuição dos dados.

Ordenação de Listas

O algoritmo realiza a ordenação de uma lista L de m números inteiros em uma máquina com n processadores.

Na **versão compartilhada**, inicialmente cada processador ordena individualmente um trecho de m/n elementos da lista e espera em uma barreira. No passo seguinte $n/2$ processadores intercalam os trechos da lista ordenados no passo anterior. Quando os $n/2$ processadores atingem uma barreira, é realizado o mesmo passo de intercalação com $n/4$ processadores e este prossegue até que toda a lista seja intercalada por um único processador.

Na **versão distribuída** a lista é dividida em m/n partes e distribuída pelos processadores. Cada processador ordena a sua lista. Em seguida, cada par de processadores intercala suas listas formando uma única lista ordenada, que é mantida em apenas um dos dois processadores. A intercalação é, então, realizada sucessivamente em pares de processadores que mantêm listas, até que um único processador mantenha toda a lista ordenada.

4.2 Medidas

Num sistema distribuído o custo de se trocar mensagens entre processadores tem peso fundamental no desempenho de uma aplicação. Para ambas as aplicações usadas em nossos testes, estamos medindo o número de mensagens e a quantidade de bytes trocados entre os processadores do sistema.

Realizamos nossos testes com 2, 4 e 8 processadores. As matrizes usadas têm dimensões 256x256 e 512x512 e a lista a ser ordenada contém 4096 e 16384 elementos.

Pela tabela 1 podemos notar que em relação à versão distribuída da aplicação de multiplicação de matrizes, o modelo *Release* gerou uma quantidade grande de mensagens, em compensação a quantidade de bytes trocados ficou bastante parecida (às vezes até menor). As mensagens geradas correspondem basicamente à leitura final da matriz resultado. Um único processador requesita a todos os outros os blocos que compõem a matriz C e para cada uma das requisições são geradas várias mensagens de controle, conforme descrito na seção 3. A aplicação de ordenação de

		Versão Compartilhada		Versão Distribuída	
Matriz	Procs	Num Msgs	Qtd Bytes	Num Msgs	Qtd Bytes
256x256	2	8194	131 072	2	262 144
	4	12 294	196 608	4	262 144
	8	14 350	229 376	8	262 144
512x512	2	32 770	524 288	2	1 048 576
	4	49 158	786 432	4	1 048 576
	8	55 467	917 504	8	1 048 576

Table 1: Multiplicação de Matrizes no Simulador do Modelo *Release*

		Versão Compartilhada		Versão Distribuída	
Lista	Procs	Num Msgs	Qtd Bytes	Num Msgs	Qtd Bytes
4096	2	2052	24 576	4	32 768
	4	5138	61 440	16	65 536
	8	8760	104 448	56	114 688
16 384	2	8196	98 304	4	131 072
	4	20 498	251 520	16	262 144
	8	34 872	417 792	56	458 752

Table 2: Ordenação de Listas no Simulador do Modelo *Release*

		Versão Compartilhada		Versão Distribuída	
Matriz	Procs	Num Msgs	Qtd Bytes	Num Msgs	Qtd Bytes
256x256	2	14	1 048 608	2	262 144
	4	32	3 145 848	4	262 144
	8	68	7 340 312	8	262 144
512x512	2	14	4 194 336	2	1 048 576
	4	32	12 583 032	4	1 048 576
	8	68	29 360 408	8	1 048 576

Table 3: Multiplicação de Matrizes no Simulador do Modelo *Entry*

		Versão Compartilhada		Versão Distribuída	
Lista	Procs	Num Msgs	Qtd Bytes	Num Msgs	Qtd Bytes
4096	2	14	65 576	4	32 768
	4	38	295 092	16	65 536
	8	96	918 064	56	114 688
16 384	2	14	131 112	4	131 072
	4	38	1 179 828	16	262 144
	8	96	3 670 576	56	458 752

Table 4: Ordenação de Listas no Simulador do Modelo *Entry*

listas apresentou resultados parecidos conforme mostra a tabela 2. A leitura final da lista por um único processador também gerou uma grande quantidade de troca de mensagens de controle entre os processadores.

O protocolo utilizado no simulador do modelo *Release* é baseado no sistema DASH, que é uma implementação em hardware de um sistema de memória compartilhada distribuída. Na definição do protocolo não houve grande preocupação em se reduzir o número de mensagens de controle trocadas entre processadores. Em termos do desempenho global da aplicação, devemos considerar que estas mensagens de controle são geradas pelos processadores em paralelo e que em geral não contêm nenhuma informação.

Os resultados do simulador do modelo *Entry* são apresentados nas tabelas 3 e 4. Devido ao protocolo gerar mensagens apenas na execução de um *acquire*, o número de mensagens trocadas entre os processadores para ambas as aplicações ficou comparável ao da verão distribuída. Já a quantidade de bytes trocados ficou bem maior que a da versão distribuída nas duas tabelas. Isto se deve à implementação das barreiras descrita na seção 3. De acordo com o modelo *Entry* os dados compartilhados são associados à barreira e cada vez que um conjunto de processos atravessa a barreira esses dados são enviados ao gerente.

O sistema Midway utiliza um mecanismo de *timestamp* para evitar essa grande quantidade de bytes trocados entre os processadores. Esse mecanismo garante que são enviados de um processador para outro apenas os dados que contêm as atualizações mais recentes. Para determinar quais atualizações são mais recentes que as outras, o sistema Midway impõe uma ordem parcial nas atualizações dos dados compartilhados relativos a determinada variável de sincronização.

5 Conclusões

Neste trabalho são discutidos os resultados preliminares de uma avaliação dos modelos de consistência fraca *Release* e *Entry* num sistema de memória compartilhada distribuída . As implementações DASH orientada para hardware e Midway orientada para software, foram selecionadas para analisar experimentalmente os desempenhos dos modelos *Release* e *Entry*, respectivamente. Os resultados são contrastados com os obtidos em um sistema de memória distribuída, no qual não há mensagens de controle e a quantidade de troca de mensagens é explícita e determinada pelo programador.

Na implementação DASH, o numero de bytes transmitido é equivalente ao do sistema de memoria distribuída, entretanto, a quantidade de mensagens , incluindo as de controle, é tremendamente maior. Em parte devido à superposição das mensagens de controle e de dados, esse overhead na prática, quando suportado pelo hardware como no projeto DASH, é tolerável e viabiliza um sistema de memória compartilhada distribuída.

Na implementação Midway simplificada utilizada neste estudo, os resultados indicam que a quantidade de mensagens trocadas fica próxima a do sistema de memoria distribuída enquanto que o número de bytes transmitido é **n** vezes superior, onde **n** é o número de processadores envolvidos. Uma implementação completa do Midway, com o uso de *timestamps* poderia eliminar transmissões redundantes e provavelmente se aproximaria do números apresentados pelo sistema de memória

distribuída. Por outro lado, uma das conseqüências da maior flexibilidade do sistema Midway é aumentar a complexidade de sua programação.

A implementação DASH do modelo *Release* embora relativamente mais eficiente é custosa em termos de projeto de hardware e seu desempenho varia com a aplicação. A implementação de uma versão preguiçosa, tal como a *Lazy Release* [10], eliminaria muito das mensagens de controle e poderia ser uma solução atrativa. Porém o uso de *timestamps* em LR, como no caso de Midway, poderá adicionar dificuldades a um projeto de sistema de memória compartilhada distribuída.

Nos próximos trabalhos pretendemos utilizar os simuladores numa maior e variada classe de aplicações e avaliar soluções híbridas. Vamos simular implementações preguiçosas em software desses modelos de consistência e que contarão com suporte de hardware.

Referências

[1] Adve, S.V. and Hill, M.D., "Weak Ordering - A New Definition", *Proc of the 17th Annual Int. Symp. on Computer Architecture*, May 1990, pp 2-14.

[2] Adve, S.V. and Hill, M.D., " A Unified Formalization of Four Shared-Memory Models", *IEEE Transactions on Parallel and Distributed Systems*, June 1993, pp 613-624.

[3] Bershad, B.N. and Zekauskas, M.J, "Midway: Shared-Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors", Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[4] Bershad, B.N., Zekauskas, M.J and Sawdon, W.A. "The Midway Distributed Memory System", *COMPCON* 1993.

[5] Carter, J.B., Bennet, J.K. and Zwaenepoel, W. "Implementation and Performance of Munin", *Proc of the 13th ACM Symp. on Operating Systems Principles*, October 1991, pp 152-164.

[6] Dubois, M. and Scheurich, C., "Memory Access Dependencies in Shared-Memory Multiprocessors", *IEEE Transactions on Computers*, June 1990, pp 660-673.

[7] Dubois, Wang, J.C., Barroso, L.A., Kangwoo, L. and Chen, Y-S,"Delayed Consistency and Its Effects on The Miss Rate of Parallel Programs", *Supercomputing* 1991.

[8] Gharachorloo, K., Gupta A. and Henessy, J.L. "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors",*Proc of the 4th Int. Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp 245-257.

[9] Gharachorloo, K. Lenoski, D., Laudon, J., Gibbons, P., Gupta A. and Henessy, J.L. "Memory Consistency and Event Ordering in Scalable Shared-Memory

Multiprocessors",*Proc of the 17th Int. Symp. on Computer Architecture*, May 1990, pp 148-159.

[10] Keleher, P., Cox, A. and Zwaenepoel, W., "Lazy release Consistency for Software Distributed Shared Memory", Technical Report TR91-168 Computer Science Department, Rice university, November 1991.

[11] Kontothanassis, L.I., Scott, M.L. and Bianchini, R., "Lazy Release Consistency for Hardware-Coherent Multiprocessors", Technical Report 547, University of Rochester, December 1994.

[12] Lamport, L., "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", *IEEE Transactions on Computers*, September 1979, pp 691-692.

[13] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta A. and Henessy, J.L. "The Directory-Based Cache Coherence Protocol for The DASH Multiprocessor", *Proc of the 17th Int. Symp. on Computer Architecture*, May 1990, pp 148-159.

[14] Singh, J.P., Weber, W-D and Gupta, A." SPLASH: Stanford Parallel Applications for Shared-Memory", *Computer Architecture News*, March 1992, pp 5-44.

[15] Zucker, R.N. and Baer, J-L, " A Performance Study of Memory Consistency Models", *Proc of the 19th Int. Symp. on Computer Architecture*, May 1992, pp 2-12.