Mário Donato Marino

Construção e Avaliação Comparativa de Um Sistema DSM

Tese apresentada à Escola Politécnica da
Universidade de São Paulo para a obtenção
do título de Doutor em Engenharia

Área de Concentração: Sistemas Digitais
Orientador: Prof. Dr. Geraldo Lino de Campos

São Paulo
Fevereiro de 2001

Marino, Mario Donato
Construção e Avaliação Comparativa de um Sistema DSM, São Paulo, 2001, 153p.
Tese (doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.
1. Sistemas DSM. 2. Sistemas Distributed Shared Memory. 3. Sistemas de Memória Compartilhada Distribuída. I. Universidade de São Paulo, Escola Politécnica, Departamento de Engenharia de Computação e Sistemas Digitais.II. t.

Ao meu avô e à minha sogra, onde quer que eles estejam.

Agradecimentos

Primeiramente, agradeço a **Deus** pelo incentivo, direcionamento e pelas oportunidades que ele tem me proporcionado neste longo caminho que percorremos.

Agradeço a minha avó, Caterina, pelo apoio e acompanhamento que tem me dado desde os meus primeiros passos. Aos meus pais, Mário e Anna, por terem me tornado esta pessoa que sou.

Agradeço ao prof. Geraldo Lino de Campos, pelos conselhos, pela disponibilização do ambiente de trabalho e pelos recursos necessários para o término deste trabalho.

Agradeço à profa. Líria, por toda ajuda na parte técnica, isto é, pelas conversas, sugestões, reflexões e auxílio sempre espontâneos.

Também gostaria de agradecer aos constantes incentivos do amigo e prof. João José Neto.

Agradeço a todos os meus colegas, em especial aos, Li Kuan Ching, Hélio Crestana Guardia e Gisele Craveiro pela convivência, auxílio, conversas e discussões.

Finalmente, gostaria de demonstrar minha imensa gratidão a minha noiva, Maria Amélia, pela paciência, compreensão e carinho em todos os momentos difíceis que passamos durante este trabalho.

Resumo

A programação de um sistema de memória distribuído é difícil pois envolve conceitos de origem, destino, empacotamento e desempacotamento de mensagens. No que tange à programação e uso da memória, um sistema paralelo de memória compartilhada é uma extensão do modelo de memória tradicional com um processador, portanto não existe a necessidade de se preocupar para onde e o que transferir de um lugar para o outro. Assim, para tornar os *clusters* de máquinas tão fácil de ser programado como uma máquina de memória fisicamente compartilhada, utilização dos clusters, a abstração de *Distributed Shared Memory* (DSM) foi criada: permite a visão de um sistema distribuído como se tivesse uma memória compartilhada única, com vários processadores (nós da rede). Para manter a visão da memória compartilhada única entre todos os nós da rede, é necessária a manutenção da consistência. O grande problema da consistência é o grande número de mensagens para mantê-la, aumenta a sobrecarga, diminuindo assim o desempenho.

Pela utilização de modelos de memória fracos, como o modelo de consistência preguiçosa e o modelo de consistência de entrada, consegue-se reduzir a quantidade de dados e mensagens que trafegam na rede para a manutenção da consistência.

Neste trabalho apresenta-se um novo sistema DSM: o *Nautilus*. O Nautilus apresenta as seguintes características: *multithreaded*, não utiliza o sinal SIGIO para a indicação de chegada de mensagens, utiliza a consistência de entrada para a diminuir a quantidade de mensagens trafegadas na rede, técnica de múltiplas escritas concorrentes para minimizar o falso compartilhamento, utiliza protocolo UDP para minimizar os *overheads* de rede e apresenta primitivas compatíveis com outros DSMs (podendo outros programas escritos em outros DSMs ou sistemas paralelos serem facilmente portados). Numa outra fase, também são avaliadas as técnicas de agregação de páginas e detecção de escrita.

Os resultados experimentais mostram que as técnicas empregadas na construção do Nautilus permitiram a obtenção de melhores tempos de execução e menor número de mensagens que outros DSMs conhecidos na comunidade acadêmica.

Abstract

It is difficult to program in a distributed system since it evolves the concepts of origin, destiny, to pack and to unpack the messages. By concerning in programming and using the memory, a shared memory parallel system is an extension of the traditional memory model with only one processor, thus there is no need of worrying about where and what to transfer from one to another place. So, the abstraction of Distributed Shared Memory (DSM) was created to become the clusters of machines to be as easily programmable as the shared memory machines: there is only one view of the memory with several processors (nodes of the network). In order to maintai the uniform view of the memory among the several nodes of the network, the consistency needs to be maintained. The immediate consequence of maintaining the consistency is the high amount of network messages, which causes a great overhead, slowing down the performance.

By using weak memory models, such as lazy release and entry consistency, it is possible to reduce the amount of data and the number of messages through the net to maintain the consistency.

In this work a new DSM system is presented: *Nautilus*. It presents the following features: multithreaded, it does not use the SIGIO signal to indicate the arrival of the network messages, it uses the entry consistency to minimize the amount of messages through the net, multiple writer techniques to minimize the false sharing effect, it uses the UDP protocol to minimize the network overheads and it has primitives compatible with other DSMs (to minimize the job of program porting). Also, other techniques such as page aggregation and write detection are evaluated.

The experimental results show that the techniques used in Nautilus construction permit to get better execution times and a lower number of messages than other well-known DSMs in academic community.

Sumário

Lista de Figuras

Lista de Tabelas

Glossário

***acquire*:** primitiva que indica ao processador (ou nó) o começo de uma seção crítica.

atualização: mecanismo que mantém coerente uma área de memória compartilhada entre vários processadores, através do envio de mensagens que contêm as modificações (por um ou mais processadores, através de semáforos) de dados de uma área de memória compartilhada.

coerência: do *cache* trata da manutenção da coerência de dados entre várias cópias de dados que podem residir nos vários *caches* e na memória principal. Um protocolo de coerência de *cache* é de fato um mecanismo de propagação de novos valores escritos de forma que todos os processadores apresentam uma visão coerente da memória compartilhada.

COMA: Cache Only Memory Access.

***copyset*:** uma lista de nós que têm uma cópia de uma página no estado *read-only* (no estado leitura).

diff*:** codificação das modificações sofridas por um processador (ou nó) durante uma seção crítica. Vem do Inglês: *difference*. Quando qualquer página for modificada (estado "leitura-escrita"), é criada, antes da modificação, uma cópia da página (twin***). Comparando-se a cópia original com a *twin* de uma página, palavra por palavra, e codificando essa comparação, temos o *diff* de uma página.

diretórios: para sistemas com barramento comum e com um número elevado de processadores, pode-se utilizar protocolos baseados em diretórios, que não requerem broadcast e consomem apenas uma fração da largura de banda do sistema. Os protocolos de diretório mantêm um índice para cada linha de memória (matriz de colunas e linhas) para ter controle de todos os processadores que fazem *cache* da linha.

***dirty bits*:** bits de rascunho, ou seja, bits para indicar que a página foi utilizada.

***false_sharing*:** quando dois ou mais processadores compartilham um mesmo bloco (página) de memória, escrevendo no mesmo, porém em variáveis diferentes.

invalidação: mecanismo que mantém coerente uma área de memória compartilhada entre vários processadores, através do envio de mensagens conhecidas como mensagens de invalidação, que ao serem recebidas pelos processadores indicarão que a área de memória compartilhada enxergada por eles não estará mais válida para efeitos de coerência; para isto, ao fazerem um próximo acesso à área compartilhada, estes processadores deverão buscar uma cópia válida da área em outros processadores a fim de poderem trabalhar com o valor correto.

lazyness: Quando se referir ao termo *lazyness* no texto, está se referindo ao atraso (relaxamento) da operação em questão. Um exemplo de *lazyness* na propagação de dados é a utilização dos

diffs no mecanismo de *multiple writer protocols* (técnica que permite que vários nós escrevam numa mesma página ao mesmo tempo).

lock-based: todas as ações de coerência são tomadas quando são feitos os acessos às *write-notices* mantidas no *lock*.

NUMA: Non Uniform Memory Access.

***polling*:** técnica que permite a consulta sem bloqueio de variáveis de estado ou portas a fim de verificar mudanças de estado.

***release*:** primitiva que indica ao processador (ou nó) o final de uma seção crítica.

seção crítica: trecho de programa, cujo acesso somente é permitido a um processador (nó de rede) por vez. O controle do trecho de programa é feito por meio de primitivas de acesso: *lock* e *unlock*.

snoopy: protocolo onde todas as ações relativas a um bloco compartilhado são enviadas por *broadcast* para todos os *caches* no sistema. A coerência de dados é mantida com cada *cache* bisbilhotando (*snoopy*) o *broadcast* e tomando as ações apropriadas de acordo com os endereços presentes em sua memória *cache*.

SO: sistema operacional.

***speedup*:** quociente entre o tempo da versão seqüencial e versão paralela. São desejáveis *speedups* maiores que 1, para mais de um processador; são desejáveis *speedups* crescentes com o número de processadores disponíveis.

write-notice: *write-notice:* estrutura transmitida junto com as mensagens de consistência e que indica quais os blocos (ou páginas) modificados numa seção crítica.

Capítulo 1

Introdução

Historicamente, as máquinas de memória fisicamente compartilhada com diversos processadores representavam o ambiente de programação paralelo mais utilizado, sendo conhecido como supercomputadores paralelos. O grande problema desse ambiente era a baixa escalabilidade pela saturação do barramento que interligava os vários processadores, bem como seu alto custo de construção.

Em contrapartida, os computadores de memória distribuída, ou seja as redes de computadores, apresentavam alta latência e baixo tempo de acesso, bem como seus processadores eram de baixo desempenho[61].

Nos últimos anos, a grande evolução das redes de computadores e dos microprocessadores, e também seu barateamento, têm tornado as redes de computadores ambientes propícios para a execução de programas paralelos.

Atualmente, as redes ou *clusters* de computadores construídos a partir de *hardware* e *software* de prateleira estão se constituindo uma regra para a redefinição do modelo de supercomputação paralela[61]. A evolução das redes de computadores e dos processadores permitiu que os *clusters* sejam encarados como supercomputadores paralelos. Como exemplo dessa evolução, podem-se citar as redes Myrinet[54] e Giga-ethernet[1] que se encontram em operação e os processadores de mais de 1GHz, como o Athlon[4], já em uso.

Muitas pesquisas têm sido feitas no sentido de popularizar os *clusters* como ambiente de baixo custo e de alto desempenho para o desenvolvimento de supercomputadores, para a execução de programas paralelos[48]. Acreditando na evolução dos *clusters*, o IEEE criou o IEEE Task Force on Cluster Computing[78].

Na linha de *clusters*, em termos gerais, têm sido desenvolvidas as seguintes áreas:

- pacotes de passagem de mensagens: PVM[60]/MPI[53];
- pacotes de comunicação a nível de usuário[84], permitindo a eliminação do TCP/UDP; podem-se exemplificar, os pacotes Active Messages[2], Fast Messages[17], Gamma[13] e Unet[7] , desenvolvimentos acadêmicos e o VIA[8], a nível industrial;

- sistemas de arquivos paralelos como o PARIO[58];
- formalização de primitivas de paralelismo para *clusters* de máquinas multiprocessadas como o padrão OpenMP[57];
- sistemas operacionais modificados como o Mosix[51];
- administração de *clusters* como o Parmon[59];
- redes de PCs e estações de baixo custo como o Beowulf[79] da NASA e o NOW[74] de Berkeley;
- *benchmarks* para *clusters*[86];
- sistemas *Distributed Shared Memory*[75] ou sistemas DSM.

Este trabalho tem como ênfase a última área, que teve início com o trabalho de Li[37] em 1986. Para explicar o conceito de DSM, faz-se necessário diferenciar entre os dois modelos de programação paralela: memória compartilhada e de memória distribuída.

1.1 Sistemas DSM

O modelo de memória compartilhada é uma extensão do modelo convencional com um único processador, pelo qual, qualquer mudança em um dado compartilhado de memória se torna imediatamente visível aos outros processadores porque a memória é fisicamente compatilhada entre os mesmos. O problema deste modelo é a alta contenção que os processadores provocam para manter a coerência de memória, apresentando assim uma grande perda de desempenho com um aumento da escala.

O modelo de memória distribuída não usa a abstração de uma única memória compartilhada no sistema, sendo que os processadores se comunicam por meio de passagem de mensagens. Se o modelo de passagem de mensagens for adotado, o programador de máquinas de memória fisicamente compartilhada vai ter de mudar seu estilo de programar, sendo obrigado a se preocupar com movimentação de dados, bem como seu empacotamento e linearização. Este modelo não apresenta o problema de contenção dos processadores pelo barramento[10]. Desta forma, para se ter a facilidade de programar do modelo de programação de memória compartilhada aliada a um bom desempenho, foi proposta a abstração de um espaço de endereçamento compartilhado entre processadores sobre um *cluster* de computadores, solução denominada de sistema *Distributed Shared Memory*[75] ou DSM, que pode ser visualizado na figura 1.1. Essa abstração permite ao usuário programar um *cluster* de computadores como se fosse uma máquina de memória fisicamente compartilhada.

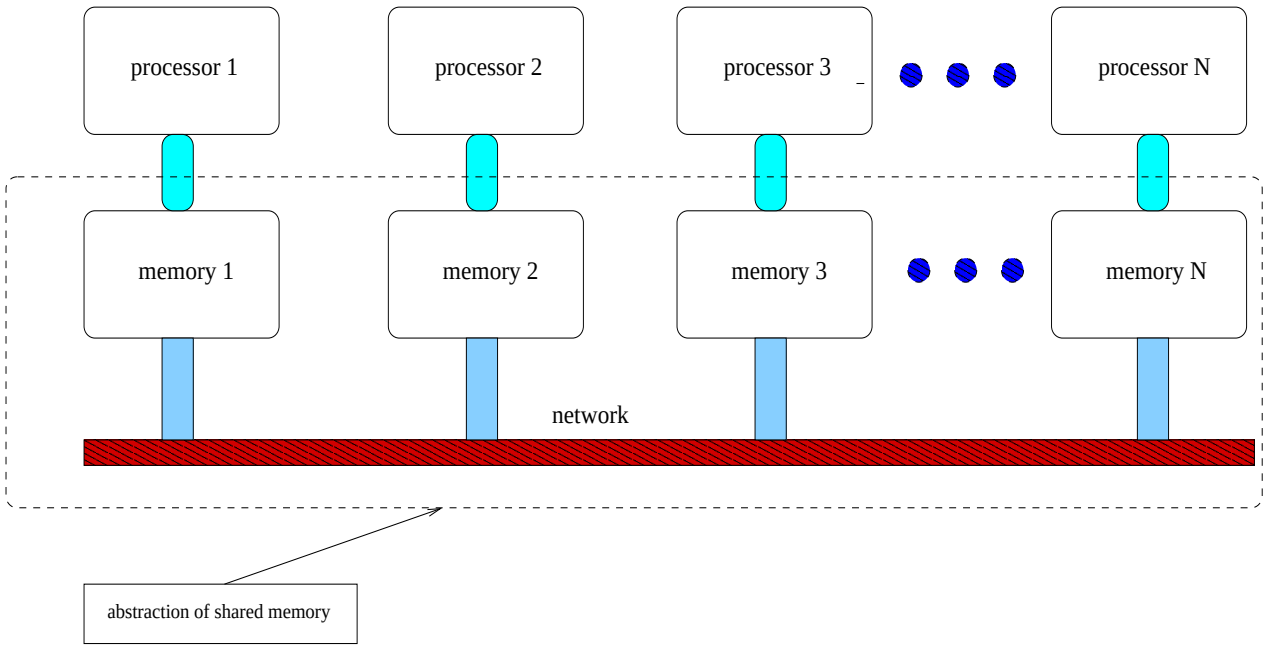

Figura 1.1: Abstração de DSM[10, 32]

Antes de explicar a constituição de um DSM, é conveniente detalhar o que são processos e *threads*[77]. Processos são entidades formadas por um espaço de endereçamento, um contador de programas, um *stack* e um conjunto de registradores. Um processo, também chamado de processo pesado, pode conter vários processos leves (*threads*), que possuem seus próprios contadores de programa e *stacks*, porém compartilham o mesmo espaço de endereçamento.

Pode-se dizer que um sistema DSM é formado vários conjunto de processos (ou de *threads*, se a implementação ou o sistema permitir), cada conjunto sendo executado em um dos nós da rede. Os processos (ou *threads*) pertencentes ao mesmo conjunto comunicam-se e os conjuntos de processos, que são executados em nós diferentes da rede, também se comunicam. O DSM deve transformar acessos à memória compartilhada em comunicação entre os conjuntos de processos[48].

Os DSMs, de forma tão transparente quanto possível, interceptam os acessos do usuário às memórias remotas e os transformam em mensagens apropriadas, que passam pela rede de interconexão. Portanto, o programador fica com a ilusão de estar com um grande espaço de endereçamento, pela eliminação explícita de troca de mensagens.

De forma análoga aos sistemas de memória compartilhada física, os DSMs podem replicar os dados, melhorando a localidade temporal e espacial. Da mesma forma que os sistemas de memória compartilhada física utilizam um bloco do *cache* para unidade de coerência, os DSMs podem também utilizar páginas de memória ou mesmo objetos como sendo sua unidade de compartilhamento. A grande unidade de compartilhamento, se comparada ao tamanho do bloco dos *caches* dos sistemas de memória compartilhada física, associada à latência dos *softwares* se combinam para constituir o grande desafio de projeto dos sistemas DSM[48].

Para melhorar seu desempenho, sistemas que apresentam memória compartilhada física utilizam *caches* locais, e portanto, replicam os dados nele contidos. Porém, ao utilizarem esses *caches*,

devem manter sua coerência, já que os dados são replicados.

Segundo Shi[66], a consistência e a coerência estão fortemente relacionadas e as condições de coerência consideram diferentes eventos de diferentes processadores para a mesma localidade, enquanto o modelo de consistência não somente considera os diferentes eventos na mesma localidade mas também impõe restrições na ordem dos eventos, isto é, na ordem de execução de cada processador. Assim uma combinação de coerência de *cache* e ordenação dos eventos irá determinar o comportamento global da memória do sistema. A seguinte relação lógica é igualmente proposta em [66]:

modelo de consistência de memória = protocolo de coerência +
ordenação dos eventos em cada processador

Na definição acima proposta, o objetivo do protocolo de coerência é garantir a visão coerente da localidade de memória dos múltiplos processadores; a ordenação dos eventos descreve a ocorrência da seqűência dos eventos de memória emitidos através de cada processador. Em resumo, um modelo de consistência de memória especifica os requisitos de coerência num protocolo de coerência, isto é, qual a visão coerente da memória compartilhada que um protocolo de coerência deve prover[19]. Um protocolo de coerência de *cache* é de fato um mecanismo de propagação de dos valores recentemente escritos na memória de forma que os processadores devam ter uma nova visão coerente da mesma[19]. Para manter a coerência, dois mecanismos básicos são utilizados: a invalidação ou a atualização.

Um modelo de memória especifica quando as operações de coerência ocorrem e quais dados devem ser visíveis. Isto pode ser implementado com vários graus de *lazyness* na propagação e na aplicação [25]:

- das operações de coerência;
- das modificações de dados que devem ser aplicadas aos nós remotos por meio da emissão de mensagens.

O que se objetiva é maximizar a localidade, minimizando ao máximo a emissão de mensagens. É por isso que os pesquisadores investiram tanto nos modelos de consistência nos últimos anos .

O *modelo seqűencial* é o que apresenta consistência mais forte, o que implica em que todas as vezes que uma variável for modificada em um nó, as modificações dessa variável se tornam imediatamente visíveis aos outros nós (através do envio de mensagens). Esse imediatismo implica em um grande número de mensagens durante a execução do aplicativo sobre o DSM, causando um baixo desempenho[10].

Se a consistência seqűencial for mantida somente nos pontos de sincronização dos programas, o tráfego de mensagens de consistência será menor e, por conseguinte, o desempenho será melhor. Os modelos de consistência que se enquadram nestas características são os modelos de *consistência de liberação* (ou *release consistency*), que somente mantêm a consistência do programa aplicativo nos pontos de sincronização, em geral barreiras e semáforos[32].

Basicamente a evolução dos sistemas DSM não é nada mais do que a evolução da *lazyness* dos protocolos. Maior *lazyness* implica em[11]:

- comunicação menos freqüente e menos operações de protocolo de rede;
- maior dificuldade de programação e maior número de estados do protocolo de manutenção da coerência.

1.2 Resumo dos Objetivos dos Sistemas DSM

Segundo [15] apud [48], o desempenho dos DSMs depende de:

- velocidade de atendimento de um pedido à memória de um nó remoto;
- velocidade com que o sistema de memória virtual detecta uma falta de página;
- *overhead* do protocolo utilizado no DSM para atender uma falta de página;
- *overhead* do *software* de comunicação e seus vários níveis de envio;
- velocidade do meio de comunicação;
- número de mensagens contendo informações de consistência, dados e sincronização;
- sincronização de barreiras e semáforos;
- balanceamento de carga.

Resumindo, um software DSM tem como objetivos principais[10]:

1. manter a coerência de memória, assim como nos sistemas fisicamente compartilhados;
2. manter a compatibilidade do *software*;
3. ser transparente ao usuário.

Sumarizando as vantagens de se programar no ambiente de um DSM[10], têm-se:

1. a não preocupação com a movimentação e linearização de dados;
2. não preocupação com primitivas do tipo *send/receive*;

3. compatibilidade com programas escritos para máquinas de memória compartilhada;

4. apresentar desempenho compatível com os mesmos programas escritos para passagem de mensagens.

Alguns DSMs podem ser citados da literatura[31]: ADSM[50]da UFRJ; Brazos[72, 73] de Rice; Cashmere[12] de Rochester; CVM[33] de Maryland; DIPC[15] de IUST; JIAJIA[20, 68, 69] (CAS); Larchant [36]; Millipede [25] (Technion) apud [14]; Mirage [49](UC Riverside); Quarks[9, 76] (Utah); SHRIMP[70] (Princeton); SVMLib[65] (RWTH Aachen), TreadMarks[32](Rice); Wind Tunnel[85] (Wisconsin); Munin[10] (Rice) e o Midway[6] (CMU).

1.3 Objetivos do Trabalho

A primeira parte deste trabalho é estudar as técnicas e modelos que os vários sistemas DSM existentes utilizam para a redução do número de mensagens e obtenção de bons *speedups*.

Na prática, para se avaliar o desempenho na área de DSMs, existem basicamente duas formas:

- simulações;

- implementação e execução de um protótipo.

De forma geral, na maioria dos artigos sobre DSMs, as comparações são baseadas em simulações, ao invés da comparação da execução de *benchmarks* em DSMs. Acredita-se que a comparação de execuções de programas sob DSMs é mais precisa e mais correta do que comparação de simulações.

A execução de DSMs em diversas redes e com diferentes sistemas operacionais pode levar a conclusões não precisas. Se vários DSMs, utilizados na comunidade acadêmica, forem executados em uma mesma rede, de prateleira, utilizando um conjunto de PCs com sistema operacional, também de prateleira, poderão ser feitas avaliações e comparações justas e homogêneas.

Um dos objetivos deste trabalho é a experimentação e comparação de vários DSMs; para isto é necessário ter-se um ambiente constituído por máquinas padrão (conjunto de PCs) que é um ambiente confiável, homogêneo e justo, para tirar conclusões a respeito do desempenho do comportamento de diferentes DSMs.

Em seguida propõe-se a construção e avaliação do software DSM denominado de **Nautilus**[1][42] em uma rede *fast-ethernet* de PCs com Linux 2.x. Este novo DSM será experimentado e comparado com outros DSMs bastante conhecidos pela comunidade acadêmica: o TreadMarks[32] e o JIAJIA[20].

[1] em homenagem ao submarino Nautilus, da obra Vinte Mil Léguas Submarinas, de Júlio Verne.

1.4 Características principais do Nautilus

O Nautilus[39, 42] é um sistema DSM que apresenta as seguintes características:

- consistência de escopo (ou *scope consistency*[26]);
- técnica de múltiplas escritas concorrentes do Munin[10];
- minimização da criação de *diffs*[26, 32, 72, 20];
- primitivas compatíveis com o TreadMarks, Quarks e JIAJIA;
- implementado para rede de PCs com Linux 2.x;
- utiliza protocolo UDP para a minimização de *overhead*s;
- *multithreading* para minimizar o chaveamento de contexto;
- não utiliza o sinal SIGIO, normalmente utilizado na implementação de DSMs para indicar a chegada de uma mensagem, diminuindo o número de chamadas de sistema e, desta forma, minimizando o *overhead*.

1.5 Justificativa

Em geral, pode-se dizer que as justificativas de se criar um novo sistema DSM são:

1. adquirir experiência prática na implementação, de forma a propor novas soluções, que na prática melhorem seu *speedup* de forma a divulgar ainda mais seu uso;
2. plataforma própria de avaliação sem a necessidade de estudo dos fontes de outros DSMs, nem sempre disponíveis;
3. avaliar outros DSMs na prática e não por simulações, já que o desempenho prático pode ser bastante diferente do simulado, se não forem levados todos os detalhes em conta, inclusive os de implementação;
4. os mesmos programas fonte podem ser usados no novo DSM, uma vez que não há necessidade de mudanças significativas, já que as mensagens envolvidas (no DSM) não são geradas de maneira explícita.

1.6 Contribuições do Nautilus

O Nautilus é o primeiro sistema DSM *multithreaded* implementado em plataforma Linux que utiliza o modelo de consistência de escopo. Esta frase é válida porque é o único DSM que simultaneamente apresenta estas características, já que:

- existem versões *multithreaded* do TreadMarks para Linux, porém ele utiliza o modelo de consistência de liberação preguiçosa, e não o modelo de consistência de escopo;
- o JIAJIA[20] é um DSM baseado na consistência de escopo porém não é *multithreaded*;
- o CVM[33] é um sistema DSM *multithreaded*, porém utiliza o modelo de consistência de liberação preguiçosa e não o modelo de consistência de escopo*;*
- o Brazos[72, 73] é um sistema DSM baseado em consistência de escopo, é *multithreaded*, porém somente opera em Windows NT, não em Linux, e assim, não pode ser adequadamente comparado.

Para melhorar o *speedup* das aplicações, duas técnicas são utilizadas pelo Nautilus:

- implementação *multithreaded*;
- *diffs* de páginas escritas pelo nó *home* não são criadas, pois estes nós já vão conter as páginas atualizadas.

A implementação *multithreaded* do Nautilus permite:

- a minimização de chaveamento de contexto;
- a não utilização do sinal SIGIO. A implementação *multithreaded* permite a eliminação do *overhead* do sinal SIGIO e a ativação do seu respectivo *handler* em todas as chegadas de mensagem.

A maior parte de sistemas DSMs baseados em páginas criados até hoje e implementados em plataformas Unix utilizam o sinal de SIGIO para ativar um *handler* que toma conta das mensagens que chegam da rede. Alguns exemplos de DSMs que se utilizam deste sinal são o TreadMarks, o Quarks e o JIAJIA. No Nautilus, um dos *threads* permanece bloqueado, quando tenta ler mensagens da rede. Enquanto bloqueado, permanece dormindo, não consumindo CPU. Esta técnica diminui o *overhead* do DSM permitindo que a CPU dê a maior prioridade possível ao programa do usuário. Portanto, o Nautilus é o primeiro sistema DSM, baseado em consistência escopo e em páginas, que não utiliza o SIGIO em sua implementação. A não utilização do sinal SIGIO minimiza a utilização da ativação de um *handler*, minimizando assim o *overhead*.

Da mesma forma que os outros sistemas, o Nautilus está preocupado em minimizar com os protocolos de rede. Para isto, o protocolo UDP é utilizado.

Como a idéia do Nautilus é aproveitar os programas utilizados pelos outros DSMs, preocupou-se com o aspecto compatibilidade. Como resultado, não existe nenhuma necessidade de rearranjo de código, quando se portam programas DSMs escritos com as primitivas do TreadMarks e do JIAJIA.

1.7 Contribuições Complementares deste Trabalho

Este trabalho apresenta várias contribuições complementares:

- primeiro DSM baseado em páginas a não utilizar o sinal SIGIO[42];
- avaliação da técnica de agregação de páginas para DSMs *home-based*[40] ;
- aplicação da técnica adaptativa de detecção de escrita CO-WD [18, 23, 45, 46] (*cache only write detection*) ao Nautilus;
- avaliação da aplicação das duas técnicas, agregação de páginas e detecção de escrita CO-WD, ambas aplicadas ao mesmo tempo ao Nautilus[46];
- iniciativa local de criar um laboratório para avaliação comparativa entre DSMs;
- metodologia prática para avaliação de DSMs.

Capítulo 2

Conceitos Básicos

O objetivo deste capítulo é estudar os tópicos mais relevantes para a análise, projeto e implementação de um sistema DSM. Vistos os conceitos básicos pertinentes à área, será dado o panorama geral atual do estado da arte na área de DSMs[25].

Os parâmetros importantes para a implementação de um sistemas DSM são apresentados a seguir.

2.1 Coerência e Consistência

Um protocolo de coerência de *cache* é um mecanismo de propagação dos valores recentemente escritos na memória de forma que os processadores mantenham uma visão coerente da mesma[19].

Censier e Feautrier [14] apud [66] definiram um esquema de coerência como coerente se o valor retornado num LOAD é sempre o valor especificado pelo último STORE com o mesmo endereço.

Esta definição aparentemente é vaga e simplista. Num sistema de computadores onde um STORE pode se armazenado em um *buffer*, não é claro se o último *store* se refere a execução do STORE por um processador ou se a um *update* da memória. De fato, a definição acima contém dois aspectos comportamentais diferentes da memória. O primeiro, chamado coerência define quais valores pode ser retornados por uma operação de leitura. O segundo aspecto, chamado ordenação dos eventos em cada processador, determina quando um valor escrito irá ser retornado por uma operação de leitura. A coerência garante que vários processadores tenham uma visão 'coerente' da mesma localidade de memória, enquanto que a ordenação dos eventos descreve quando os outros processadores vêem um valor que foi atualizado por um processador. Em [38] apud [66], Henessy e Patterson apresentaram condições suficientes para coerência como se segue:

1. uma leitura por um processador P em uma localidade X, que segue uma escrita por P a X, com nenhuma escrita de X por nenhum outro processador entre a escrita e a leitura por P, sempre retorna o valor escrito por P;

2. uma leitura por um processador em uma localidade X, que segue uma escrita por um outro processador em X, retorna o valor escrito da leitura se a leitura e a escrita são suficientemente separadas e desde que nenhuma outra escrita em X ocorra entre os dois acessos;

3. escritas a uma mesma localidade são serializadas, isto é, duas escritas a uma mesma localidade por dois processadores quaisquer são vistas na mesma ordem por todos os processadores.

Segundo Shi [66], as três condições acima somente garantem que todos os processadores tenham uma visão coerente sobre a posição X. Portanto, não informam sobre a ordenação dos eventos em cada processador, e não se pode determinar quando um valor escrito pode ser visto pelos outros processadores. Desta forma, a fim de capturar o comportamento do sistema de memória precisamente, é necessário impor a coerência e a ordenação dos eventos, que é a regra do modelo de consistência de memória.

A consistência e a coerência estão fortemente relacionadas. A coerência considera diferentes eventos de diferentes processadores para uma mesma localidade, enquanto o modelo de consistência não somente considera os diferentes eventos numa mesma localidade, mas também impõe restrições na ordem dos eventos, isto é, na ordem de execução de cada processador. Assim uma combinação de coerência de *cache* e ordenação dos eventos irá determinar o comportamento da memória inteira do sistema. É proposta a segunte relação lógica por Shi em [66]:

modelo de consistência de memória = protocolo de coerência +
ordenação dos eventos em cada processador

Podem-se definir modelos relaxados de consistência de memória, ou modelos de consistência fraca como modelos onde somente a ordem em cada processador é relaxada, em sucessivas operações de sincronização[66]. Entre os modelos de consistência, a ordenação dos eventos em cada processador é uma característica similar entre eles, e a principal diferença entre os vários modelos é o método de manutenção da coerência, o que inclui dois aspectos: qual protocolo de coerência é utilizado e como este protocolo é implementado[66].

Em resumo, um modelo de consistência de memória especifica os requisitos de coerência num protocolo de coerência, isto é, qual a visão coerente da memória compartilhada que um protocolo de coerência deve prover[19]. Um protocolo de coerência de *cache* é um mecanismo de propagação dos valores recentemente escritos na memória de forma que os processadores tenham uma visão coerente da mesma[19].

2.2 Classificação de Protocolos de Coerência

Na classificação proposta por Hu em [19], vários fatores foram considerados:

- para quem o novo valor escrito na memória deve ser propagado: *snoopy*, diretórios e *lock-based*;

- como propagar o novo valor escrito na memória: servidor central (*central server*), migração (*migration*), replicação de leitura (*read replication*) e replicação plena (*full replication*);

- quem pode escrever o novo valor em um bloco: múltiplos protocolos de escrita (*multiple writer protocols*) e protocolo simples de escrita (*single writer protocol*);

- quando propagar o novo valor escrito: propagação adiantada ou atrasada.

Os dois primeiros aspectos são expostos a seguir. Os outros aspectos serão vistos quando forem discutidos os modelos de consistência.

2.2.1 Protocolos snoopy e de diretórios

Em protocolos *snoopy* todas as ações relativas a um bloco compartilhado são enviadas por *broadcast* para todos os *caches* no sistema. A coerência de dados é mantida com cada *cache* bisbilhotando (*snoopy*) o *broadcast* e tomando as ações apropriadas de acordo com os endereços presentes em sua memória *cache*. Com a facilidade do *broadcast*, somente as ações da controladora de *cache* local e a informação dos estados distribuída localmente são necessários para manter a coerência. Protocolos *snoopy* são idealmente adequados para multiprocessadores que utilizam um barramento do tipo compartilhado, pois um barramento compartilhado provê um *broadcast* rápido e barato, mas que pode se tornar um gargalo quando o número de processadores aumenta.

Para sistemas com barramento comum e com um número elevado de processadores, pode-se utilizar protocolos baseados em diretórios, que não requerem *broadcast* e consomem apenas uma fração da largura de banda do sistema. Os protocolos de diretório mantêm um índice para cada linha de memória (matriz de colunas e linhas) para ter o controle de todos os processadores que fazem *cache* da linha. Esta informação é então usada para seletivamente invalidar/atualizar sinais quando uma linha de memória é escrita. Protocolos de diretório são adequados para multiprocessadores com redes gerais de interconexão. Sistemas de memória compartilhada com coerência de *cache* baseada em diretórios podem ser escalados até milhares de processadores[19]. A desvantagem dos protocolos baseados em diretórios é que à medida que o número de processadores aumenta, a complexidade de manter a coerência também aumenta, sendo necessária mais memória para manter os diretórios. Além disso, a complexidade de manter as entradas do diretório consistentes com as cópias *cache* limita a escalabilidade[19].

2.2.2 Protocolo de Coerência *Lock-based*

No trabalho de Hu[19], propõe-se o protocolo de coerência *lock-based* onde se elimina totalmente o diretório. A idéia básica deste protocolo é que todas as ações de coerência são tomadas quando são feitos os acessos às *write-notices* mantidas no *lock*.

Este protocolo adota a organização NUMA, onde a memória compartilhada é distribuída entre os nós, que desta forma são *homes* de algumas páginas. Segundo este protocolo, uma página pode estar no estado de leitura, leitura-escrita ou invalidado. Qualquer processador no sistema pode fazer acessos a partes remotas da memória compartilhada através da rede de interconexão. Quando uma falha de página ocorre, o dado necessário é buscado, de acordo com seu endereço físico, da memória local ou remota[19]. Quando ocorrer um *release* de um *lock*, todos os *diffs* produzidos nas seções críticas serão enviados para os *homes* (proprietários das páginas).

2.2.3 Algoritmos ou Protocolos de Propagação

A classificação proposta por Stum[75], define os algoritmos ou protocolos de consistência que implementam a manutenção da consistência nos sistemas DSM. Os protocolos de propagação permitem migrar ou replicar os dados. Os que migram, exploram o princípio da localidade nos acessos. Os que replicam, permitem múltiplos acessos de leitura simultâneos, aumentando-se o paralelismo.

A classificação de Stum[75] propõe os algoritmos servidor central (*central server*), migração (*migration*), replicação plena (*full-replication*) e replicação de leitura (*read-replication*).

No algoritmo servidor central (*central server)* existe um servidor central responsável por atender os acessos de dados e por manter uma cópia única dos mesmos. A sua principal vantagem é sua implementação simples. Sua desvantagem é que se todos os clientes fizerem pedidos ao servidor, este acaba sendo o gargalo. Uma solução é ter-se vários servidores, particionando-se os dados adequadamente.

No algoritmo de migração (*migration*), os dados são migrados para os nós onde são feitos os acessos. Este algoritmo pode ser classificado como SRSW (*single reader single writer*), porque os *threads* que são executados no mesmo nó podem ler ou escrever dados a qualquer momento após a migração. Este algoritmo pode ser facilmente integrado ao sistema de memória virtual, se o tamanho do bloco de dados que for migrado for múltiplo do tamanho da página. Supondo que um bloco de dados seja igual a uma página, se o acesso for feito a um dado que não está localizado no nó local, uma falha de página é disparada. Existe então um administrador de falha de página que irá se preocupar em localizá-la em um dos nós da rede. A desvantagem deste algoritmo é que se a mesma página for utilizada por mais de um nó, a página vai ficar saltando entre os nós, fenômeno conhecido por *thrashing* ou *ping-pong*.

O algoritmo replicação de leitura (*read-replication)* pode ser classificado como MRSW (*multiple readers single writer*), pois se a leitura de um dado é necessária, é feita uma busca do bloco de dados que o contém entre os nós da rede. Sendo o bloco encontrado, é feita uma cópia para o nó local. Se for feita uma escrita no dado, todas as outras cópias do bloco de dados devem ser invalidadas. A vantagem deste algoritmo é a redução do custo das leituras devido à replicação dos blocos, pois a replicação dos mesmos permite que as leituras sejam locais e em paralelo nos vários nós. Um exemplo deste algoritmo é o sistema DSM Ivy proposto por Li[37], que utiliza o

conceito de proprietário da página, onde cada proprietário possui sua própria *copyset*. Se ocorrer uma leitura, o proprietário deve incluir o nó que pediu a página na sua *copyset*. Se ocorrer uma escrita, a propriedade da página é transferida do nó proprietário para o nó onde a falta de página (por escrita na mesma) ocorreu e o proprietário antigo envia para todos os nós da sua *copyset* uma mensagem para que eles invalidem suas cópias locais. Em seguida, o novo proprietário muda a propriedade da página para *read-write*. Este protocolo de consistência também é bastante conhecido como *write-invalidate* ou de invalidações.

O algoritmo de replicação plena (*full-replication*) pode ser classificado como MRMW (*multiple readers multiple writer*), permitindo que blocos sejam replicados na escrita também. A manutenção da coerência de dados é mais complicada, pois não existe uma ordem que os dados devem ser mantidos[75]. Para a manutenção da ordem, pode existir um seqűenciador global, que associa um pedido de escrita de um bloco a um número; os outros nós ao receberem uma mensagem com este número, o associam à modificação do dado. Este protocolo de consistência também é bastante conhecido como *write-update* ou de atualizações.

Segundo Stum[75], para DSMs cujos blocos são organizados como páginas, o algoritmo de replicação de leitura diminui o número de buscas de páginas na leitura, dando um comportamento melhor para a maioria das aplicações, que estatisticamente apresentam alta localidade e portanto altas taxas de acertos. Ainda segundo Stum[75], o algoritmo de replicação plena é adequado quando a replicação é feita em baixa escala e quando as atualizações não são freqűentes, isto é com poucas atualizações, como por exemplo para aplicações hipermídia.

2.3 Protocolos de coerência

Existem dois tipos básicos de protocolos de coerência: *write-invalidate* e *write-update*. Os protocolos de *write-update* garantem que todos os processadores que têm cópias vejam o novo valor simultaneamente com o processador modificado, enquanto o protocolo *write-invalidate* alcança a visão coerente pela invalidação das outras cópias[19].

Nos protocolos *write-invalidate*, se um bloco for modificado por um processador, todas as cópias deste bloco nos outros processadores devem ser invalidadas. Quando um dos outros processadores quiser referenciar este bloco, uma cópia dele deve ser buscada e colocada em seu *cache*. Nos protocolos *write-update*, se um bloco for modificado por um processador, o novo valor escrito neste bloco é propagado em forma de diretórios a todos os outros processadores para que tenham o valor também atualizado[19].

Segundo Stum[75], os protocolos *write-invalidate* são bons para aplicações que apresentam um padrão seqűencial de compartilhamento, enquanto os protocolos *write-update* em diversas aplicações com diversos padrões de compartilhamento. Também ode-se aplicar uma combinação deles para obter um padrão de compartilhamento que se adeqúe a elas, proporcionando assim um melhor desempenho do sistema.

O *multiple writer protocol* é proposto para minimizar os efeitos do falso compartilhamento, que ocorre quando dois ou mais processadores concorrentemente atualizam diferentes dados compartilhados que estão localizados em um mesmo bloco. De maneira análoga, num protocolo *write-update* uma modificação em uma palavra de um bloco causa o envio do bloco ou palavra a todos os processadores que possuem uma cópia do bloco, mesmo que esses processadores não mais utilizem essa palavra[19].

2.4 Alguns Modelos de Consistência Importantes

Existe um compromisso entre a facilidade de programação e o potencial de ganho de desempenho: quanto mais fácil o modelo de consistência para o programador, menor o ganho de desempenho do modelo e vice-versa.

Esta seção está baseada nas referências [10], [48] e suas referências.

Em ordem crescente de ganho de desempenho (ordem decrescente de facilidade de programação): consistência estrita, consistência seqűencial, consistência de processador, consistência fraca, consistência de liberação, consistência de liberação preguiçosa, consistência de escopo e consistência de entrada.

As palavras "processador" e "nó" podem ser perfeitamente intercambiáveis neste contexto.

2.4.1 Consistência estrita (Strict consistency - StC)

Neste modelo, a leitura de um endereço de memória retorna o valor que foi escrito mais recentemente nesse endereço.

2.4.2 Consistência seqűencial (Sequential consistency - SC)

O modelo de consistencia seqűencial (*sequential consistency*), implementado na maioria dos processadores *snoopy-cache*, impõe que as modificações feitas na memória compartilhada distribuída sejam imediatamente visíveis aos outros processadores, isto é, o resultado da execução de um aplicativo é o mesmo se as operações de todos os nós foram executadas em qualquer ordem seqűencial e as operações de cada nó individualmente aparecem na ordem dada pelo seu programa. Portanto, por este modelo, todos os nós devem concordar com a ordem dos efeitos observados, devendo a escrita na memória compartilhada por um nó se tornar imediatamente visível aos outros.

Segundo Dubois [23]apud [48]:

- uma leitura de um processador p_i é efetuada com relação a um processador p_j quando a emissão de uma escrita por p_j no mesmo endereço não puder afetar o valor retornado para p_i;

- uma escrita pelo processador p_i é efetuada com relação ao processador pj quando uma leitura do mesmo endereço por p_j retornar o valor definido pela escrita (de p_i);

- um acesso é efetuado quando o for em relação a todos os nós da rede;

- um acesso de leitura (primeira definição) é globalmente efetuado quando ele é efetuado e a escrita (segunda definição), onde se retorna o valor, também foi efetuada;

Para se ter consistência seqűencial precisa-se:

- antes que uma leitura seja efetuada com respeito a qualquer processador, todas as leituras prévias devem ser globalmente (em todos os processadores) efetuadas e todos os acessos de escrita anteriores devem ser efetuados;

- antes que se permita que uma escrita por um processador seja efetuada com respeito a qualquer outro processador, todas as leituras prévias devem ser globalmente efetuadas e todos os acessos de escrita anteriores devem ser efetuados.

O modelo de consistência seqűencial é simples de se implementar e garante ao programador que nenhuma inconsistência ocorra. Antes de uma escrita de um dado compartilhado ser completada, todas as cópias são atualizadas ou invalidadas. A grande desvantagem deste modelo é a latência.

A figura 2.1 mostra os problemas deste modelo. Suponha-se que os processadores p1 e p2 possuem uma cópia das variáveis x, y e z e que p1 modifique seus dados dentro de uma seção crítica. Na figura 2.1, w significa escrita e a seção crítica está delimitada pela *acquire* (aquisição) do semáforo e pela *release* (liberação) do mesmo. A escrita de Wy deve ser atrasada até que a escrita anterior Wx seja completada, mesmo numa seção crítica. A consistência seqűencial exige que p1, mesmo estando numa seção crítica se comunique com p2 (comunicação representada por flechas) sempre quando um dado compartilhado é modificado. O problema é que p1 fica parado (linhas pontilhadas) enquanto aguarda a confirmação de que p2 recebeu a mensagem de p1 (mensagem de ack, representada com linha tracejada). O grande número de mensagens e o fato de p1 ficar parado aguardando os *acks* são a razão do baixo desempenho da consistência seqűencial.

2.4.3 Consistência de processador (Processor consistency - PC)

Um sistema DSM adota este modelo quando as escritas vistas individualmente em cada nó nunca são vistas fora de ordem, mas quando vistas em dois nós, podem ser observadas de forma diferente. Formalmente:

- antes que uma leitura seja feita por um processador, todos os acessos de leituras prévias devem ser efetuados;

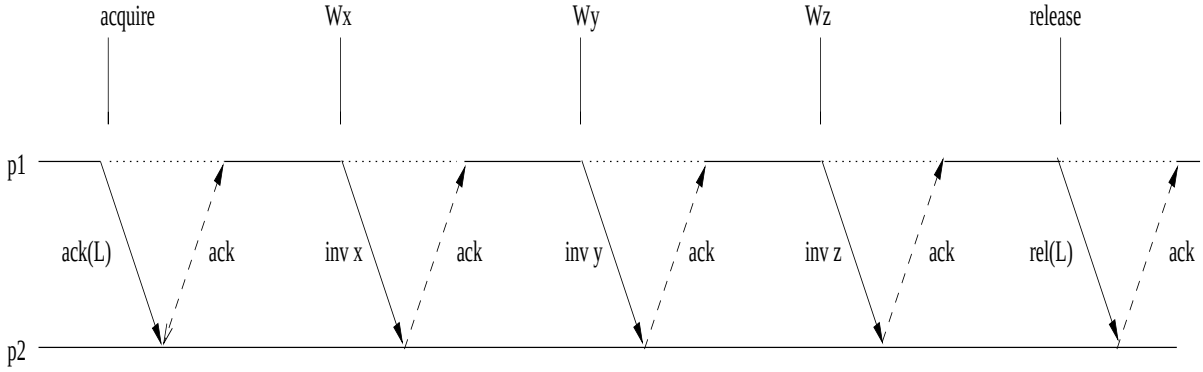

Figura 2.1: Consistência seqűencial utilizando invalidações [20, 21]

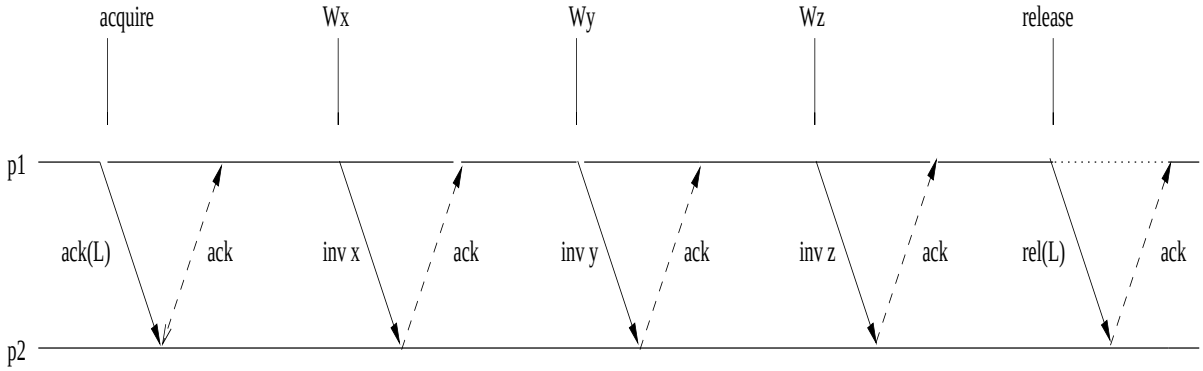

Figura 2.2: Consistência de processador utilizando invalidações[10, 11]

- antes que uma escrita seja feita por um processador, todos os acessos prévios devem ser efetuados.

Este tipo de modelo permite o *pipelining* das escritas, resultando num tempo de paralelização menor dos nós. Para manter a ordem uma FIFO é utilizada.

Um exemplo de máquina que adota este modelo são as PRAMs (*Pipelined Random Access Memory*), onde os nós da rede mantêm uma cópia completa da memória global compartilhada, lendo e atuando nas variáveis compartilhadas de sua cópia local. As atualizações para a manutenção da consistência são transmitidas mais tarde para os outros nós segundo uma FIFO, o que garante que são observadas na mesma ordem por todos os nós.

O DASH [Leno90] apud [10], adota este tipo de modelo, permitindo ao processador, que transmite as invalidações, ficar no estado *stalled* (linha pontilhada) uma só vez durante uma seção crítica (*acquire/release*), esperando pelo ack da mensagem de invalidação (em tracejado), como pode ser visto na figura 2.2. Como se pode ver por esta figura, o problema deste modelo é ainda o grande número de mensagens (de invalidação) para manter o sistema coerente.

2.4.4 Consistência fraca (Weak consistency - WC)

Este modelo de consistência impõe que o valor retornado por uma operação de leitura é o valor que foi escrito por uma operação de atualização na mesma variável, que pode ter imediatamente

precedido a leitura na execução de um processo.

A consistência é garantida pelo programador por meio de operadores de sincronização, tornando o sistema seqűencialmente consistente nos pontos de sincronização. Formalmente:

- os acessos de sincronização são seqűencialmente consistentes um com relação ao outro;
- nenhum acesso a uma variável de sincronização é emitido por um processador antes que todos os acessos anteriores tenham sido realizados;
- nenhum acesso normal, isto é, que não seja de sincronização, é emitido por um processador antes que um prévo acesso a uma variável de sincronização tenha sido feito.

Os acessos de sincronização atuam como delimitantes, de forma que no instante de sincronização todos os acessos normais anteriores tenham sido efetuados e todos os acessos normais posteriores não tenham sido efetuados. Dependendo do tipo de implementação adotado, a consistência fraca não pode ser considerada um modelo mais fraco, isto é, um modelo onde a quantidade de vezes que o sistema seja mantido coerente seja menor, pois existem ordenamentos admitidos por um e não pelo outro.

Um programa executado num modelo de consistência fraca apresenta os mesmos resultados se for executado num modelo seqűencial se não existem *data races* (disputa de dados), isto é, se na execução não existir um par de operações conflitantes sendo pelo menos uma delas a dados e não ordenadas pela relação parcial definida em [Adve93] apud [32].

As vantagens de se utilizar o modelo de consistência fraca são evitar a sincronização desnecessária, reduzindo o número de mensagens para a manutenção da consistência, de sincronização e de dados.

2.4.5 Consistência de liberação (Release consistency - RC)

O modelo de consistência de liberação, foi desenvolvido como parte da máquina DASH para minimizar o excesso de mensagens geradas na consistência seqűencial.

Acessos sincronizados são acessos que competem e são utilizados para garantir a ordem e a atomicidade entre múltiplos acessos. Acessos não sincronizados são acessos que competem, porém não garantem ordem.

Segundo o modelo de consistência de liberação, os acessos a dados compartilhados podem ser não sincronizados ou sincronizados, estes últimos divididos em *acquire* e *release*. Um *acquire* sinaliza que os dados compartilhados são necessarios, enquanto que o *release* sinaliza que os dados compartilhados estão disponíveis. Grosseiramente, os acessos *release* e *acquire* correspodem às operações de sincronização de um semáforo. **A chegada a uma barreira pode ser modelada como um release; a saída de uma barreira, modelada como um *acquire*.**

Antes de formalizar a consistência de liberação, é importante conceituar[10]:

1. um programa apresenta vários acessos de leituras e escritas;

2. acessos prévios são acessos precedentes ao corrente acesso na ordem do programa;

3. uma leitura de memória é *performed*, i.e., torna-se visível com respeito a qualquer outro processador quando uma escrita subseqűente na mesma não puder afetar a leitura;

4. uma leitura de memória é visível com respeito a qualquer outro processador quando uma leitura subseqűente retornar o valor armazenado pela escrita;

5. leituras e escritas se tornam *performed* quando são visíveis a todos os outros processadores.

Formalizando a consistência de liberação, tem-se:

- antes que uma leitura de memória ou escrita na memória possam ser executadas com respeito a qualquer outro processador, todos os *acquires* prévios devem ser executados;

- antes que um *release* com respeito a qualquer processador possa ser executado, todas as escritas na memória e leituras da mesma devem ser executadas;

- antes que um *acquire* com respeito a um semáforo ou saída de barreira possam ser executados com respeito a qualquer processador, todos os *releases* prévios devem ser executados; antes que um *release* seja visível com respeito a qualquer outro processador, todos os *acquires* e *releases* prévios devem ser executados.

Se a consistência for mantida somente nos pontos de sincronização, inconsistências temporárias vão ocorrer. Quando um processo necessita fazer um acesso a uma variável compartilhada, ele precisa fazer o *acquire* do semáforo, necessitando nesse ponto ter uma visão correta das variáveis compartilhadas. Quando se termina o acesso a uma variável compartilhada numa seção crítica, o processo deve liberar o semáforo (*release*) correspondente a essa seção crítica. Para isso, ao terminar o uso das variáveis compartilhadas, o processo que as utiliza, envia as modificações (aos outros processos que possuem uma cópia da variável compartilhada), ou *diffs*, sofridas pelas variáveis durante a seção crítica, se o protocolo de consistência for o de atualização, ou envia mensagens de invalidação aos outros processos que possuem uma cópia da variável compartilhada, se o protocolo de consistência for o de invalidação.

Pelo armazenamento das modificações em *buffers*, e pela combinação de várias mensagens contendo as modificações, ou contendo as invalidações, a serem propagadas aos outros nós da rede, a consistência de liberação consegue diminuir o número de mensagens, permitindo um aumento de desempenho sobre a consistência seqűencial e sobre a consistência de processador. A figura 2.3 exemplifica o modelo de consistência de liberação.

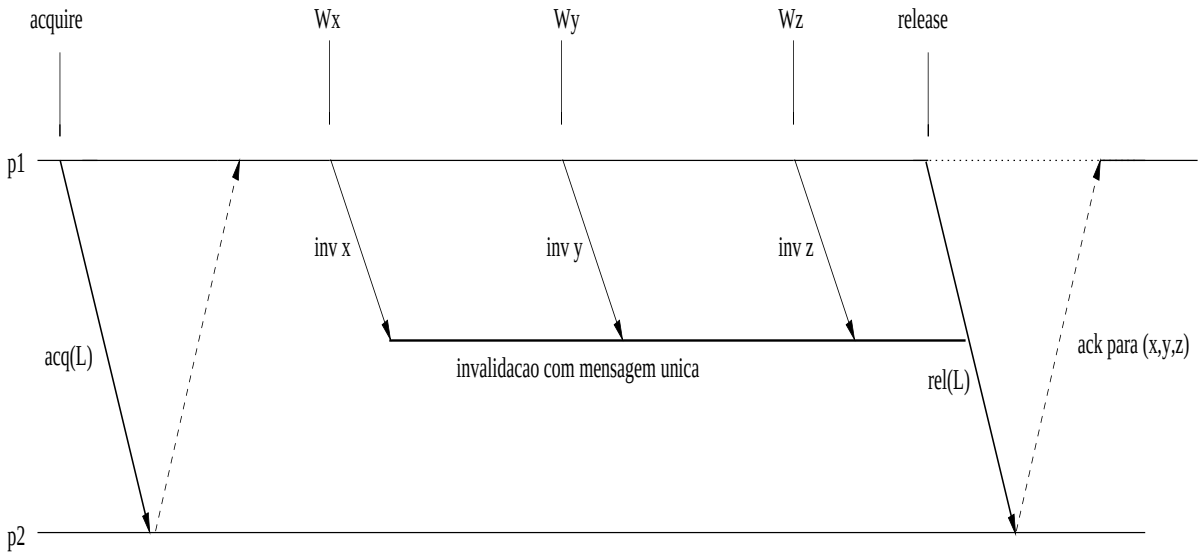

Figura 2.3: Consistência de liberação com invalidações [10, 11]

Concluindo, programas escritos para o modelo de consistência de liberação apresentam os mesmos resultados do modelo de consistência seqűencial, desde que se utilize uma sincronização adequada[10].

Na figura 2.4 apresenta-se o diagrama de estados das páginas para um DSM baseado em consistência de liberação, organizado em páginas e com protocolo de invalidação.

2.4.6 Múltiplos Protocolos de Escrita ou Múltiplas Escritas Concorrentes (*Multiple Writer Protocols*)

A técnica de múltiplos protocolos de escrita, como usada no Munin[10, 11], minimiza o efeito do falso compartilhamento. Esta técnica consiste em se salvar todas as modificações feitas pelos nós durante uma seção crítica. Além disso, esta técnica permite que vários nós modifiquem variáveis diferentes alocadas numa mesma página, daí o nome múltiplas escritas concorrentes (*multiple concurrent writers*).

Detalhando a criação dos *diffs*, como pode ser vista na figura 2.5, pode-se dizer que antes da escrita em qualquer página pertencente à memória compartilhada, as variáveis compartilhadas aí armazenadas estão no estado "somente leitura".

2.4.7 Consistência de Liberação Preguiçosa (Lazy release consistency - LRC)

Neste modelo de consistência, a propagação das modificações sofridas durante uma seção crítica é atrasada até o próximo *acquire* (saída de uma barreira ou aquisição de semáforo). Este modelo

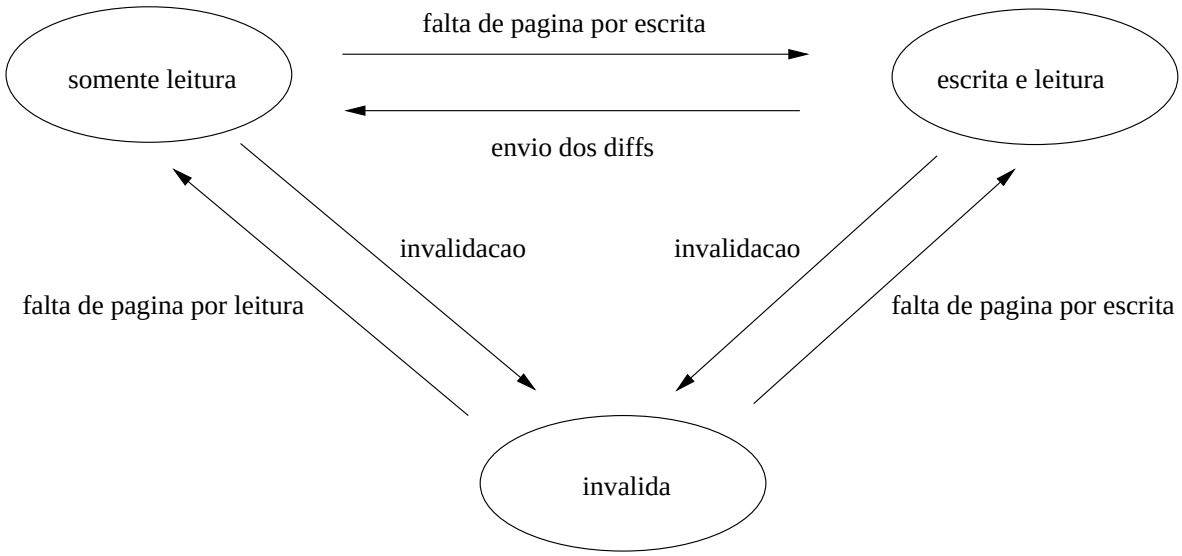

Figura 2.4: Diagrama de estados das páginas de um DSM baseado em consistência de liberação com invalidações [10, 11]

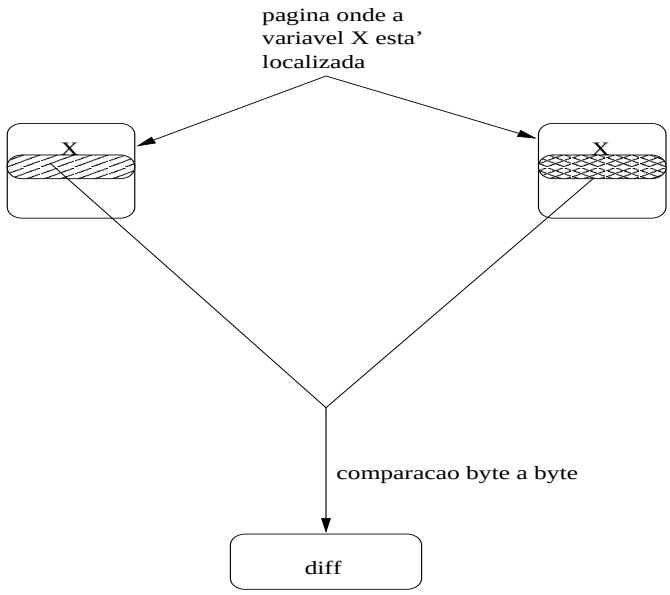

Figura 2.5: Criação de *diffs* [10, 11]

procura reduzir o número de mensagens emitidas e a quantidade de dados trocada não propagando as modificações no momento do *release*. Formalmente [32]:

- antes que um acesso de leitura ou escrita possa ser visível com respeito a qualquer outro processo, todos os *acquires* devem ser vistos, com respeito a qualquer outro processo;
- antes que um *release* possa ser visto com respeito a qualquer outro processo, todas as leituras e escritas anteriores devem ser vistas com respeito a qualquer outro processo;
- acessos sincronizados são seqűencialmente consistentes entre si.

No modelo de consistência de liberação, a propagação das modificações sofridas por uma página durante uma seção crítica são atrasadas até o próximo *release*, quando são transmitidas a todos os nós que possuem uma cópia da mesma. Porém, pode ser que alguns nós não mais a utilizem, sendo, portanto, uma mensagem de modificação transmitida inutilmente. No modelo de consistência de liberação preguiçosa os nós que irão precisar da página adquirem o semáforo e buscam as modificações desejadas[48].

A execução em cada nó é dividida em intervalos de tempo (*time-stamps*), cada um representando uma seção crítica. Estes intervalos são denotados por índices de tempo, ordenados pelo relógio da máquina. Entre processadores diferentes, se o índice for maior que o índice local, significa que foram feitas modificações mais recentes, e portanto a variável está desatualizada, sendo necessária a aplicação das modificações.

Segundo o modelo *lazy*, quando se executa um *acquire*, o nó que faz o *release* envia seu índice de tempo junto com a mensagem de *acquire*, indicando quais as páginas modificadas (chamadas de *write-notices*) em índices mais recentes que o atual. Se um nó receber um *write-notice* referente a uma página, ela será invalidada, de tal forma que, será necessário trazer uma nova cópia atualizada da mesma. Se a página já existir localmente, devem-se buscar as modificações que a tornem atual, isto é, são determinados os *diffs* e o conjunto de nós de onde se devem buscá-los, devendo estes ser aplicados de acordo com os índices de tempo crescentes. Dessa forma, o modelo *lazy* procura sempre fazer trafegar *diffs* ao invés de páginas inteiras, o que minimiza a quantidade de dados, diminuindo o *overhead* do DSM. Como os *diffs* somente são criados quando pedidos por um nó, menos *diffs* são criados desnecessariamente (o que é conhecido como *lazy diff creation*), minimizando-se os *overheads*[32].

Como se pode perceber, este modelo é bastante eficiente, porém não só o mecanismo de obtenção dos *diffs* é complicado, mas também, exige que se guarde todos os *diffs* de todas as seções críticas, o que acaba causando a ocupação de uma grande quantidade de memória, o que, em certas condições[41, 47], pode causar o fenômeno de *swapping*.

Na figura 2.6 pode ser visto um exemplo de consistência relaxada preguiçosa.

Neste modelo, pode-se utilizar protocolos de consistência baseados em invalidações, atualizações e até híbrido.

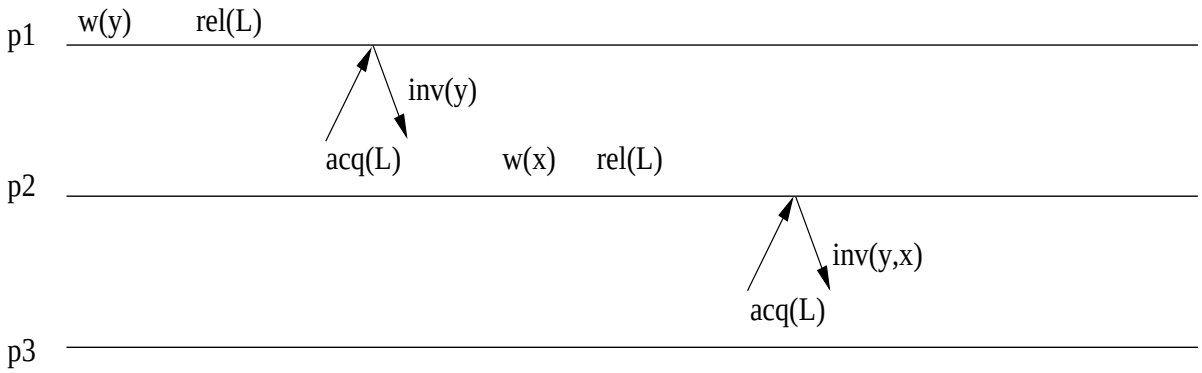

Figura 2.6: Exemplo de consistência relaxada preguiçosa[32]

Os resultados dos programas executados utilizando o modelo de liberação preguiçosa são os mesmos da consistência seqűencial, segundo a relação parcial definida em [Adve93] apud [10], desde que na execução não existir um par de operações conflitantes, sendo pelo menos uma delas a dados, e desde que não ordenadas consecutivamente.

Segundo Zhou[87], o protocolo LRC apresenta os seguintes problemas:

- o processamento, criação e a aplicação de *diffs* é um *overhead* e polui o *cache* do processador;
- o processador que faz o *acquire*, poderá visitar mais de um nó para obter os *diffs* necessários a serem aplicados na mesma página;
- mesmo que os vários *diffs* sejam obtidos de um mesmo nó, eles serão obtidos separadamente e aplicados individualmente na página. Transferir uma página, neste caso, minimiza o número de mensagens e o *overhead* de aplicação dos *diffs*;
- a memória para armazenar *diffs* e *write-notices* pode crescer rapidamente;
- o processo de *garbage collection* deve ser aplicado para reduzir a quantidade de memória consumida.

2.4.8 Home-*based lazy release consistency* (HLRC), *Automatic update release consistency* (AURC) e *Overlapped* home-*based lazy release consistency* (OHLRC)

No estudo de Zhou[87], foram introduzidos os seguintes modelos adicionais:

- o HLRC (*Home-based* LRC, isto é, um LRC baseado em *homes*);

- o AURC (*Automatic Uptade Release Consistency*), que além de ser baseado em *homes*, apresenta ajuda de *hardware* especializado, que não detecta as modificações das páginas por meio de *diffs* (em software). As modificações são produzidas numa cópia principal da página compartilhada que está num nó *home*, fazendo com que as atualizações sejam feitas por meio de uma simples cópia do nó *home*;

- o OHLRC (*Overlapped Home-based* LRC), que tira vantagem da comunicação entre processadores (como o exemplo do Paragon[87]), a fim de aliviar algumas tarefas protocolos do HLRC.

2.4.9 Consistência de Entrada (Entry consistency - EC)

O modelo de consistência de entrada (*entry consistency*) pressupõe que um dado compartilhado deve ser associado a uma variável de sincronização, isto é, toda vez que for feito um acesso a um dado compartilhado, necessariamente precisa-se ter uma variável de sincronização a ele relacionada. Logo, um *acquire* e um *release* de uma variável de sincronização que guarda o acesso à seção crítica, onde é feito o acesso ao dado compartilhado, são necessários. Neste modelo de consistência, somente os acessos às variáveis de sincronização (que guarda a seção) se tornam visíveis, isto é, o *release* não tem nenhuma função[6].

Os acessos às variáveis podem ser feitos de modo exclusivo ou não exclusivo. No modo exclusivo, leituras simultâneas podem ser feitas. Escritas somente podem ser feitas no modo não exclusivo. O comprimento da seção crítica na consistência de liberação é maior que o da consistência de entrada[6], onde os dados atualizados devem ser buscados antes que a seção crítica começa, como mostra a figura 2.7. Para impor a ordem das atualizações recentes, adota-se uma relação parcial, baseada no estudo de Lamport[Lamp78] apud [48], onde um evento A acontece antes de B se uma mensagem for enviada do nó que contém A para o nó que contém B.

Finalmente, este modelo pode ser considerado um caso extremo do modelo *lazy*, pelo fato de se associar variáveis compartilhadas a variáveis de sincronização.[6, 26, 32].

A associação de dados compartilhados à variáveis de sincronização pode não ser fácil [20] e infelizmente, modelos derivados do modelo de consistência de liberação aumentam a complexidade do modelo de programação[20].

Formalizando[6]]:

- antes de um *acquire* se tornar visível, todos as atualizações aos dados guardados devem se tornar visíveis com respeito ao processo;

- quando se é feito um *acquire* em modo exclusivo, nenhum outro processo pode fazer o *acquire* do *lock*, nem mesmo em modo não exclusivo;

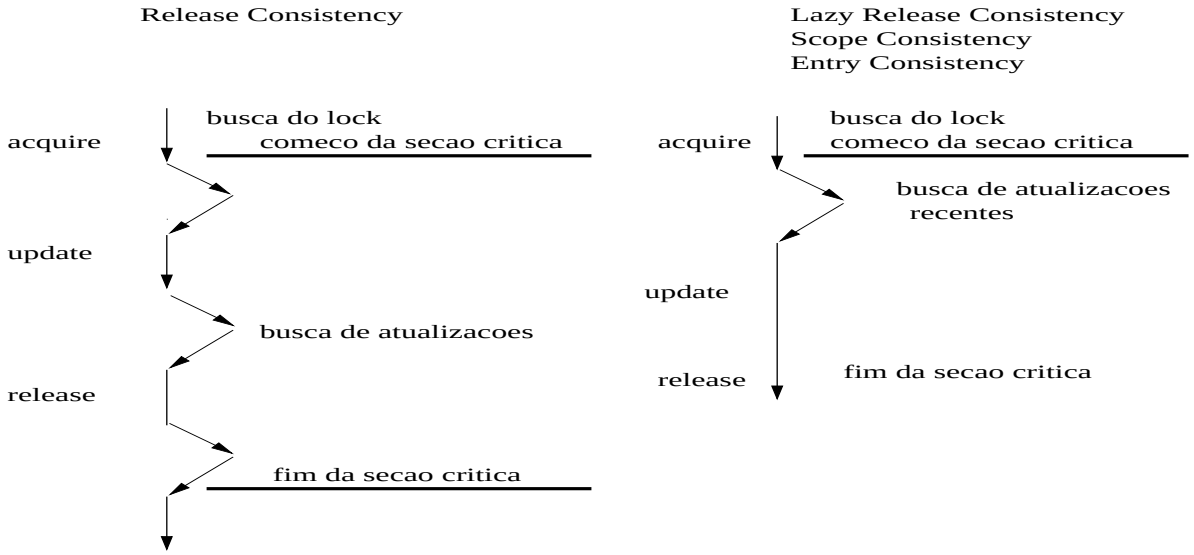

Figura 2.7: Comparação entre *release consistency*, *lazy release consistency*, *scope consistency* e *entry consistency* [6]

- depois de se fazer um *acquire* em modo exclusivo, o próximo *acquire* não exclusivo do memo *lock* deve ser visto por todos os outros processos somente depois de ser visto com respeito ao dono do *lock*.

2.4.10 Consistência de escopo (Scope consistency - ScC)

Um escopo de consistência (*consistency scope*) é uma visão limitada da memória com respeito a quais referências de memória se tornam visíveis, isto é, modificações de dados feitas dentro de um escopo de consistência são somente garantidas visíveis aos outros nós dentro daquele escopo.

Pode-se pensar um escopo de consistência como sendo todas as seções críticas protegidas pelo mesmo *lock*, ou mesmo barreiras que definem um escopo de consistência global incluindo o programa inteiro. Para cada escopo de consistência existe uma operação de abertura (*open*) e uma operação de fechamento (*close*). O intervalo durante o qual um escopo é aberto a um dado processo é chamado seção. Quaisquer modificações feitas dentro de uma seção de escopo se tornam visíveis aos processos que entram em novas seções daquele escopo. Modificações feitas fora de uma seção de escopo não têm garantia de ser visíveis pelos outros nós[20].

Formalizando[20]:

- antes que um escopo de consistência seja aberto por um processo P, todas as escritas vistas com respeito àquele escopo de consistência devem ser vistas com respeito a P;

- um acesso à memória é visto com respeito a um processo P somente depois de todas as seções de escopo que P previamente entrou serem abertas.

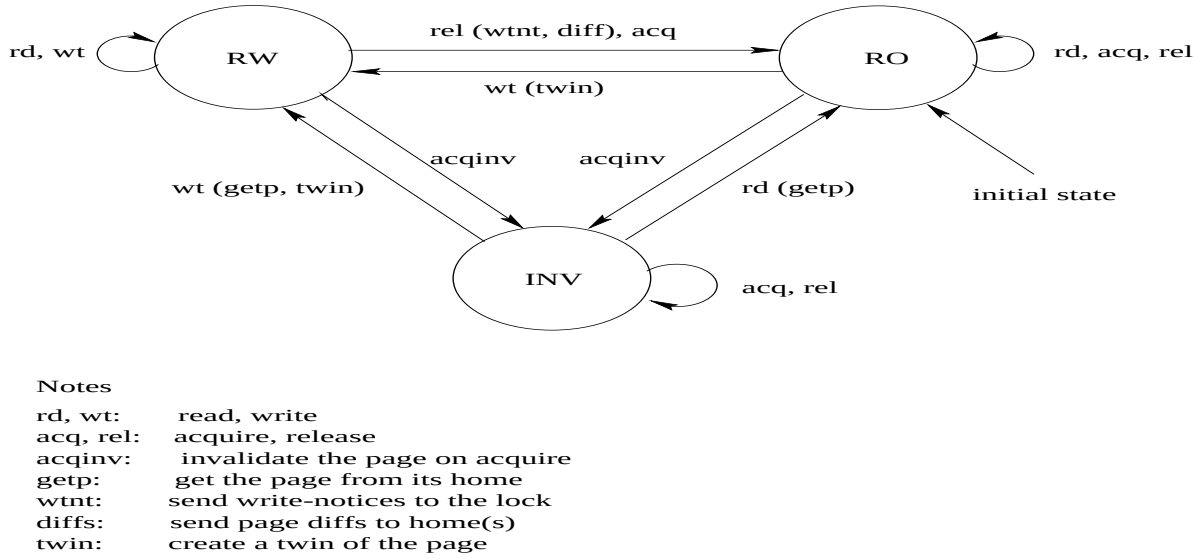

Figura 2.8: Diagrama de transição de estados do protocolo *lock-based* para a implementação da consistência de escopo[20]

A primeira regra estabelece que depois de um processo abrir um escopo de consistência, todas as atualizações que devem ser previamente vistas com respeito ao escopo são garantidas serem vistas com respeito àquele processo. Em outras palavras, as atualizações que ocorreram previamente em seções fechadas de um escopo, se vistas por um processo ou por outros, são garantidas serem vistas na memória do processo local que vai abrir uma nova seção daquele escopo[20]. Forçando as operações abertas a se completarem antes que os acessos à memória subseqűentes sejam visíveis, a segunda regra garante que os acessos a memória compartilhada sejam atualizados de acordo com a primeira.

O modelo de consistência de escopo se propõe[26] a ser uma ponte entre o modelo de consistência de entrada e o modelo de consistência de liberação. Ao invés de associar *locks* a dados compartilhados, *locks* são associados a seções críticas ou, mais genericamente, a escopos de consistência. Formalmente:

- um acesso de escrita ou leitura à memória é permitido com respeito a qualquer outro processador somente após todos os *acquires* anteriores terem sido feitos;
- um *release* é permitido a qualquer processador após todos os acessos prévios aos dados compartilhados (protegidos pelo *lock*) terem sido feitos;
- acessos de sincronização são seqűencialmente consistentes.

A maioria das aplicações escritas para os modelos de liberação e liberação preguiçosa podem ser executadas no modelo de consistência de escopo sem nenhuma ou com pequenas modificações.

Pode-se ver na figura 2.8, o diagrama de transição de estados para este modelo de consistência.

O modelo ScC oferece a maioria das vantagens do modelo EC sem requerer a associação de variáveis de sincronização a dados compartilhados. Estabelece-se o conceito de escopo, definindo uma associação transparente, não explícita entre variáveis de sincronização e dados compartilhados. Programas escritos para o modelo LRC podem ser executados perfeitamente no modelo ScC sem modificações, na maioria dos casos, isto é, seus eventos são representados pelos escopos. Porém nem todos os programas escritos para o modelo LRC são corretos, podendo haver necessidade adicional de definir alguns escopos nesses casos. Mesmo nesses casos, a extensão é facil, pois escopos simples são associados a seções de código e não a mudanças na estrutura de dados.

Nos programas com sincronização simples, os *locks e* barreiras delimitam escopos de consistência, não se necessitando a inclusão de nenhum outro mecanismo. Seções críticas protegidas pelo mesmo *lock* definem um escopo, e uma particular seção crítica protegida por aquele *lock* e que seja utilizada delimita uma seção de escopo[20]. O *acquire* do *lock* abre a seção crítica e o *release* do *lock* a fecha. Barreiras definem um escopo de consistência global, inicialmente aberto por todos os processos. Cada processo que entra na barreira fecha a seção de escopo e abre uma nova na saída da barreira. Se um processo passou por uma barreira, é garantido a ele ver todas as mudanças por qualquer processos antes da barreira, como no modelo RC[20]. Múltiplos processos podem simultaneamente fazer escritas em uma barreira, mas somente um deles pode fazer uma escrita protegida por um *lock*. Em termos de barreiras e *locks* pode-se formalizar[20]:

- antes que um *lock* ou barreira possa ser visível a um processo P, todas as escritas vistas com respeito àquele *lock* ou barreira devem tambem ser vistas com respeito a P;

- um acesso à memória é visível a P se somente a um processo P somente depois que todos os *acquires* prévios a P ocorram.

Na figura 2.9 mostra-se um exemplo de consistência de escopo.

O processo P0 faz o *acquire* do *lock* L0 e entra no escopo definido por aquele *lock*. O processo P0 escreve em X0 dentro do escopo L0 e escreve em X1 enquanto dentro dos escopos L0 e L1. Depois o processo P1 faz o *lock* L1 abrindo o escopo L1 e lendo X0 e X1. Sob o modelo de consistência de escopo para P1, garante-se ver a escrita de P0 a X1 assumindo que P1 faz o *lock* de L1 depois de P0 fazer o *release* do mesmo. Portanto, não existe a garantia de que P1 irá ver a escrita em X0 porque o escopo L0 não foi aberto por P1 quando o mesmo leu X0; assim, P0 não precisa propagar a modificação de X0 para P1 (nos modelos RC e LRC precisa-se). Portanto, se P1 fez o *acquire* de L0 ao invés de L1 e se isso aconteceu depois que P0 fez o *release* de L0, então as escritas a X0 e X1 visíveis por P0 serão visíveis a P1.

Um programa é Sc-correct se ele observar a consistência seqűencial de memória quando for executado em um sistema que suportar a consistência de escopo. Formalizando o conceito de Sc-correct[20], tem-se:

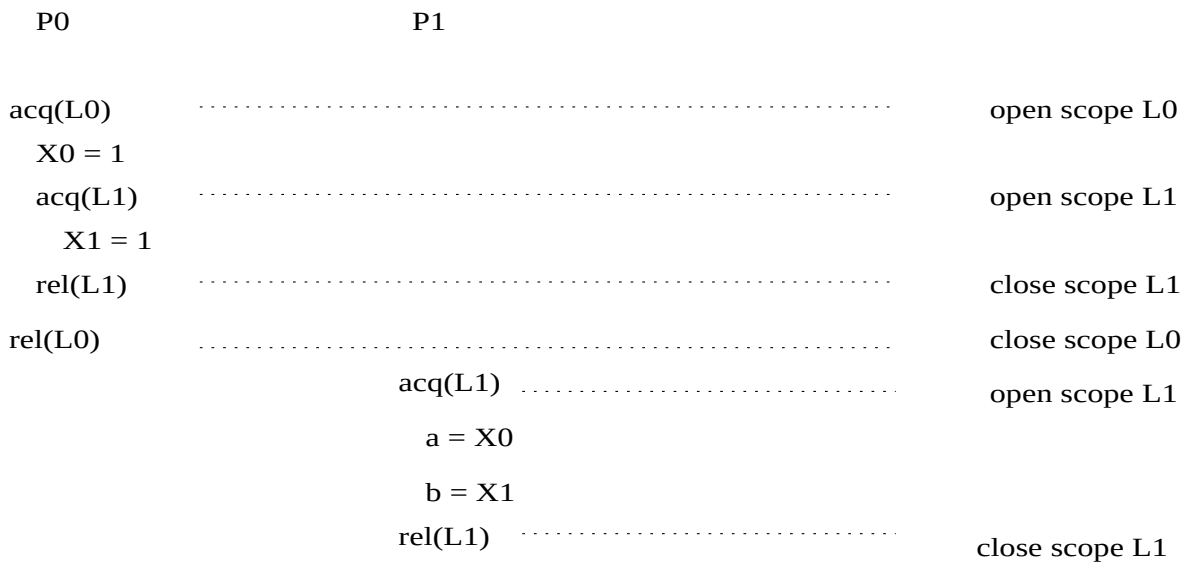

Figura 2.9: Programando com escopos *lock-based*[20]

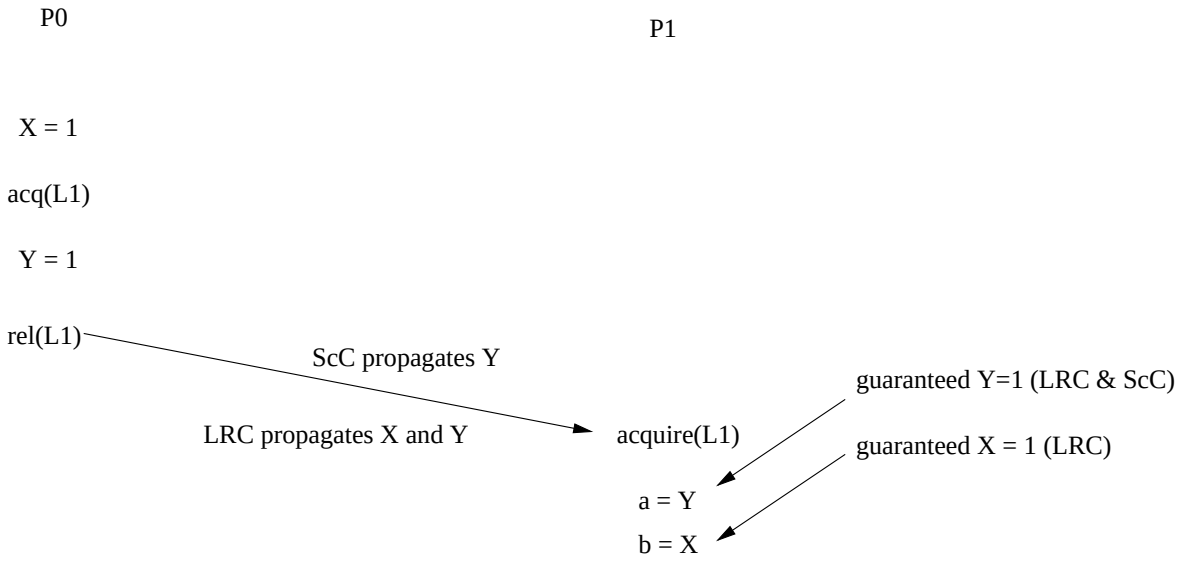

Figura 2.10: Consistência de escopo (ScC) versus consistência de liberação preguiçosa (LRC)[20]

- modificações protegidas por *locks* a dados compartilhados não são esperadas ser visíveis em um processo antes que pelo menos um dos *locks* sejam adquiridos por estes;
- modificações a dados compartilhados não protegidas por *locks* não serão visíveis aos processos antes da próxima barreira.

Na figura 2.10 mostra-se outro exemplo de consistência de escopo.

P0 escreve primeiramente em X e depois em Y; Y é escrito na seção crítica guardada pelo *lock* L1, enquanto X é escrito fora dela. P1 faz o *acquire* de L1 depois de P0 ter feito o *release*. Sob o modelo LRC, depois de fazer o *acquire* de L1, a P1 é garantido ver todas as escritas que ocorreram em P0 antes do *release* de L1, assim os valores novos de X e Y são atualizados. A consistência ScC garante somente que P1 vê o novo valor de Y, porque o escopo está sendo feito sobre o L1. Para

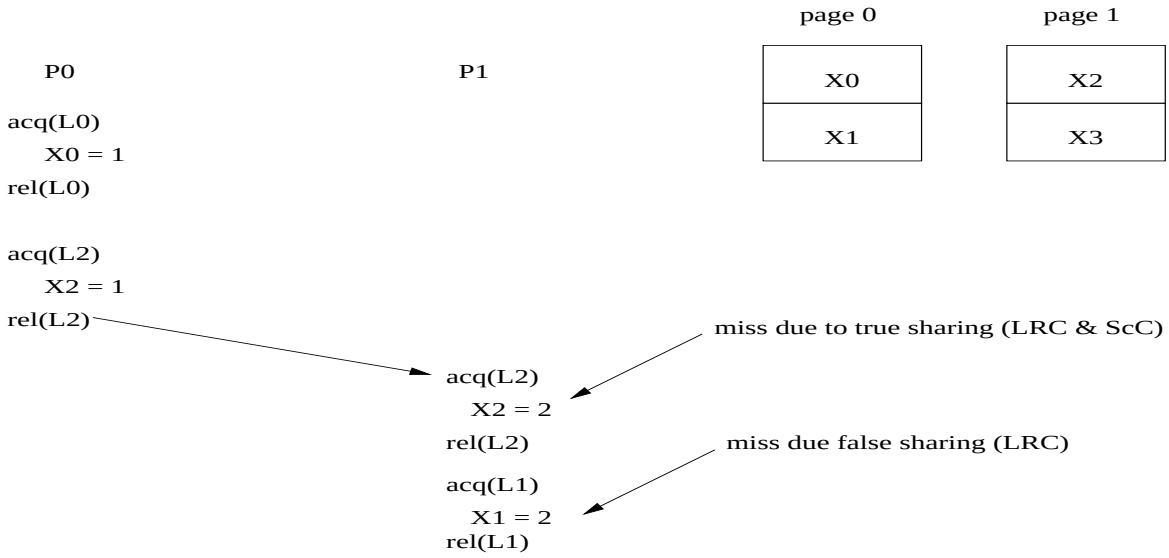

Figura 2.11: Falso compartilhamento no modelo ScC versus no modelo LRC[20]

garantir visibilidade a X e Y, a escrita em X por P0 deve estar dentro da mesma seção crítica que a em Y, caso contrário, X somente será visível a P1 na próxima barreira[20].

Programas que usam filas de tarefas para distribuir a computação exemplificam esta situação: a seção crítica protege a fila de tarefas somente e nem todas as escritas a dados são visíveis pela tarefa. Portanto, o programador assume que quando uma tarefa é selecionada da fila de tarefas, as atualizações previamente feitas pelas outras tarefas devem ser visíveis aos outros nós, que é uma afirmação correta no modelo RC, mas não correta no ScC. Somente serão válidas neste último se as modificações na fila de tarefas forem protegidas por *locks*[20]. Algumas precauções devem ser tomadas: tornar seções críticas grandes o suficiente para conter todas as modificações; adicionar novas seções críticas para proteger modificações não protegidas ou mesmo, utilizar escopos explícitos[20].

Os modelos ScC e RC não requerem associação explícita entre dados compartilhados e sincronização. O ScC assume a associação implícita entre acessos à memória a escopos determinados da estrutura do programa. O modelo ScC reduz-se ao modelo RC para aplicações que usam somente sincronização global como barreiras, porque então existe um único escopo global[20].

Pelo atraso da coerência do *release* para o *acquire* e suportando múltiplas escritas concorrentes na mesma página, o modelo LRC elimina o efeito do falso compartilhamento, comparado com o modelo RC. Mas no modelo LRC todas as modificações devem ser vistas pelo processador que fez o *release*[20].

O modelo ScC vai mais adiante na eliminação dos efeitos do falso compartilhamento. Suportando vários escopos de memória, o modelo permite que atualizações sejam postergadas até que o correspondente escopo seja aberto, permitindo que o processo que faz o *acquire* veja as modificações atrasadas que ocorreram no mesmo escopo[20].

A figura 2.11 ilustra o benefício do modelo ScC em relação ao LRC com respeito à redução do

falso compartilhamento. P0 modifica primeiro X0 e depois X2, pelo *acquire* dos *locks* correspondentes, L0 e L2. Quando P1 faz o *acquire* de L2, sob o modelo LRC ele deve ver as modificações de X0 e X2, pois o *acquire* permite que depois de todos os acessos prévios à memória sejam visíveis, e desta forma, ambas as páginas 0 e 1 serão invalidadas por P1. De fato, P1 não precisa ver a modificação a X0 agora desde que ele não fez o *acquire* em L0 e X1 deve estar atualizado. A invalidação na página 0 que ocorreu quando foi feito o *acquire* em L2 causa uma falha quando for feito um acesso a X1 (devido ao falso compartilhamento X0 e X1 residem na mesma página)[20].

No modelo ScC, quando P1 faz o *acquire* em L2, somente as modificações protegidas por L2 são propagadas. Assim uma invalidação e a subseqűente falta de página da página 1 vão ocorrer, mas não da página 0. De maneira análoga ao RC, o modelo ScC requer sincronizações explícitas (barreiras e *locks*). Garantir que um programa feito para RC esteja correto em ScC não é trivial e requer que dependências de dados sejam verificadas. Algumas modificações podem ser necessárias para essa conversão[20].

No modelo EC a associação entre objetos de dads e objetos de sincronização é especificada pelo programador. De fato, este modelo foi proposto para DSMs baseados em variáveis e não em páginas. Como as associações entre dados e variáveis de sincronização é feita de modo a identificar as modificações a serem propagadas em determinadas variáveis, este protocolo é implementado com um protocolo de atualização. No EC a associação entre dados e *locks* é explícita e no ScC é implícita, sendo neste último criada pelos acessos à memória[20].

No modelo EC a sincronização explícita é difícil[20]:

- não se associam dados a barreiras, requerendo que se utilizem *locks* para tornar as modificações visíveis após as mesmas;
- existem associações explícitas a *locks*, requerendo que múltiplas associações se barreiras separarem diferentes fases da computação;
- pode-se também fazer uma associação do espaço de endereçamento inteiro, o que gera muita comunicação num protocolo baseado em atualizações como o Midway[6].

No modelo EC a granularidade é do tamanho da variável, pois dados são associados a variáveis de sincronização. A granularidade é um fator importante no desempenho e na complexidade do programa. Se as associações foram feitas com pedaços grandes e contínuos de memória o efeito do falso compartilhamento pode ser reintroduzido e *locks* adicionais são necessários. Associações de dados pequenos não contíguos são mais eficientes porque requerem menos comunicação e processamento, porém são mais difíceis de serem programadas, já que o programador deve estabelecer mais associações para cada *lock*. A grande tarefa do programador é escolher uma boa associação. A comunicação depende de como as modificações forem detectadas. O Midway[6] utiliza *dirty bits* em software, assim a quantidade de comunicação é diretamente proporcional ao tamanho dos dados a serem atualizados.

No modelo ScC a associação é determinada pela granularidade, dada pelo tamanho da palavra. Qualquer modificação que ocorre num escopo automaticamente associa a correspondente localização daquele escopo. Portanto a granularidade de comunicação típica do modelo ScC é o tamanho de página se o modelo ScC for implementado com um protocolo de memória virtual compartilhada, transferindo páginas contendo os dados. Embora o volume de comunicação possa ser maior no ScC do que no EC devido a fragmentação, a transferência de um bloco de tamanho igual ao de uma página pode beneficiar o modelo ScC com técnicas de *pré-fetching*[20, 80].

O modelo ScC é potencialmente mais fácil de ser programado do que o EC não requerendo a mudança em muitos casos de programas escritos para RC. Quando programas RC não são ScC, anotações devem ser adicionadas e seriam semelhantes a adições de explícita associação em programas EC, embora seja muito mais fácil por ser orientado a código do que a dados.

2.4.11 Vantagens e Desvantagens dos Protocolos *Home*-based

Nos DSMs tipo *home-based*, as páginas apresentam proprietários fixos. À medida que são necessárias nos outros nós da rede, as páginas são replicadas. Para manter a coerência, nos nós onde existem cópias são criados os *diffs*, que por sua vez são enviados para seus respectivos *homes*.

Um mesmo modelo de consistência pode ser implementado como *home-based* ou *homeless*.

Baseados no estudo de Keleher[34, 35], os protocolos LRC *home-based* podem apresentar desempenho melhor que os protocolos LRC *homeless* para alguns tipos de aplicações e os principais limites dos protocolos *home-based* são relacionados a adaptação de padrões de compartilhamento.

São vantagens dos protocolos *home-based*[62, 34, 35]:

- modificações feitas nos nós *home* não requerem a criação de *diffs*;
- para atualizar a página, somente é necessário um par de mensagens (pedido/resposta);
- os estados das páginas não são persistentes não se necessitando o processo de *garbage-collection*.

São desvantagens[34, 35]:

- os nós *homes* devem ser escolhidos com cuidado;
- a comunicação entre dois nós é feita utilizando o nó *home*, pois os *diffs* são nele aplicados e depois a página atualizada deve ser transmitida;
- *diffs* são criados em cada intervalo e não atrasados, como nos protocolos tipo *lazy*;
- através de uma mensagem que contém uma página, nós diferentes do *home* podem atualizar suas páginas.

2.5 Técnica de Agregação de Páginas

Na maioria dos DSMs que utilizam uma organização estruturada por páginas, existe um *handler* responsável por pedir uma página a um outro nó quando uma falta de página ocorre. Sempre que é feito um acesso a uma página que não está com permissão nem de leitura nem de escrita ou uma página que está com permissão de leitura sofrer uma escrita, um sinal SIGSEGV é gerado e o *handler* mencionado anteriormente é ativado para pedir a página. Este *handler* pode mudar a permissão da página para *read-only* ou *read-write*. No Unix, esta mudança de permissão é feita através da primitiva *mprotect()*.

A primitiva *mprotect()* permite considerar a granularidade da região a ser modificada como sendo múltipla do tamanho de uma página. Isto permite modificar a permissão de mais de uma página ao mesmo tempo, ou seja, pode-se enchergar a uma página, que tradicionalmente apresenta o tamanho 4kB como tendo tamanho múltiplo deste, isto é: 8kB, 16kB, etc. Esta capacidade de considerar granularidades múltiplas de 4kB, tamanho tradiconal de página, é conhecida como técnica de agregação de páginas.

Considerando que a maior parte dos DSMs implementados em forma de biblioteca usam o UDP, que possui tamanho de buffer máximo limitado a 64kB, para não ter perda de desempenho empacotando e desempacotando mensagens, os DSMs que utilizam esta técnica de agregação limitam o tamanho da página em 32kB.

No estudo de Amza[5], chega-se à conclusão que aplicando-se a técnica de agregação, o falso compartilhamento é incrementado, porém o número total de mensagens trocadas é reduzido. E também, caso o nó faça o acesso de várias páginas (de 4kB) sucessivamente, um único pedido e resposta pode ser suficiente ao invés de vários pedidos e respostas (com página de tamanho menor), desta forma, menos mensagens relativas a páginas. O estudo de Amza [5] também mostra que existe uma redução do número de falhas de página, mas o falso compartilhamento pode aumentar a quantidade de dados e o número de mensagens trocadas.

2.6 Outros Aspectos Relevantes

2.6.1 Tipo de Mensagem

As estatísticas do comprimento da mensagens[22] apud [67] nos sistemas DSM mostram que o padrão de tráfego é do tipo bimodal com duas principais categorias: longas e curtas. Mensagens longas incluem páginas, grandes *diffs* e resposta de sincronizações; mensagens curtas envolvem pedidos de sincronização, pedidos de páginas e pedidos por *diffs*. O tamanho das mensagens curtas é menor que 20 bytes e o tamanho das mensagens longas depende do aplicativo[67]. Dependendo do tamanho da mensagem, a taxa de transferência muda, e por isto, é bastante importante conhecer o comportamento do aplicativo[13], isto é, qual a percentagem de mensagens pequenas e grandes.

2.6.2 Obtenção de *diffs*

Várias mensagens são geradas para a obtenção dos *diffs* aumentando a latência quando ocorre uma falta de página em um modelo que trabalhe desta forma. Além disto, a necessidade de armazená-los pode gerar um grande uso de memória e conseqüentemente exigir um processo de *garbage collection* (o que pode gerar um *overhead* substancial). Mais ainda: os *diffs* se acumulam rapidamente, já que não podem ser descartados, uma vez que podem ser necessários par um futuro nó que fizer o *acquire*. Além disso, para gerar *diffs*, é intenso o uso de operações de memória:

- criação da página *twin*;
- comparação palavra por palavra para calcular o *diff*;
- aplicação do *diff* para a atualização de páginas;
- poluição do *cache* primário e diminuição do desempenho[20] apud [25].

Nos protocolos baseados em *homes* com mecanismos de múltiplas escritas concorrentes:

- menos mensagens de faltas de páginas vão ocorrer, isto é, para se obter uma página basta trazê-la, ao invés de buscar vários *diffs*;
- os *diffs* somente são aplicados nos nós *home* da página;
- escritas nos nós *homes* não geram *diffs*;
- não existem buscas de página no nó *home*;
- se uma única palavra de uma página for escrita, a página inteira será buscada (na falta de página);
- se os nós *homes* não forem bem escolhidos, o desempenho poderá ser baixo, dependendo do aplicativo.

2.6.3 Protocolos Adaptativos

Uma outra forma de aplicação da *lazyness* na aplicação dos dados é constituída pelos protocolos adaptativos, que mudam a técnica de *multiple writer protocols*, para *single writer protocol*, isto é, utilizam esta mudança quando ocorre um comportamento migratório. Protocolos *home-based* tratam de situações com múltiplas escritas de maneira mais eficiente do que protocolos adaptativos[5, 23, 18].

Outras formas alternativas de controlar a propagação de dados é a inclusão de alguns bits por palavra, inclusão esta sendo feita em *software* [4] apud [25]. Para padrões migratórios, esta implementação comporta-se melhor do que a utilização de *diffs*, que por sua vez se comportam melhor a outros padrões devido ao custo dessa instrumentação [1] apud [25].

Outros trabalhos como os de Wu[18, 21, 23, 83], através de protocolos adaptativos aplicados ao JIAJIA[20] melhoraram o desempenho em até muitos pontos percentuais, dependendo do programa aplicativo avaliado.

2.6.4 Comunicação a nível de usuário

Pacotes de comunicação a nível de usuário diminuem o *overhead* pela não utilização das camadas de comunicação típicas, e também são colocados no mesmo espaço de endereçamento do usuário, o que por sua vez, reduz o número de trocas de contexto e o número de chamadas de sistema. Alguns pacotes de comunicação a nível de usuário têm sido utilizados pela comunicade acadêmica: Active Messages[2], Fast Messages[17] e Unet[7] . O VIA[8] é uma versão comercial do Unet que já foi padronizada e está começando a ser utilizada.

2.6.5 Interrupção ou Polling

Nenhuma conclusão foi obtida até hoje, isto é, se é melhor a utilizar interrupção ou *polling*. O que foi concluído é que, dependendo da situação, um ou outro pode ser mais adequado.

2.6.6 Latências

Para minimizar o problema da latência, os pequisadores têm proposto com as técnicas de *prefetching* e *multithreading*[34, 41] apud [80]. As técnicas de *prefetching* propostas nas simulações de Bianchini[4] apud [25] e o *multithreading* dos sistemas Brazos[72] e CVM[33] têm mostrado bom desempenho.

2.6.7 Compilador

O projeto Midway[6] que propôs a *entry consistency* foi abandonado desde 1993. Este projeto foi um dos poucos a se preocupar com a implementação do DSM orientada a compilador.

Outros projetos como o CAS-DSM[56] retomaram recentemente estas idéias adotando através de instrumentação de código e uma cuidadosa análise feita em tempo de compilação, através da eliminação do sinal de violação de proteção de segmentação (SEGV) e seu respectivo *handler*. No trabalho de N.P.[56], pela instrumentação do código e optimizações agressivas do compilador SUIF aplicadas ao CVM[33], bons níveis de *speedup* foram atingidos.

Capítulo 3

Principais Sistemas DSM

Este capítulo envolve o estudo das principais características de sistemas DSM existentes e propõe uma classificação em três gerações de sistemas DSM. Esta classificação é diferente das propostas de Carter[9] e Speight[72].

Carter propôs uma classificação de DSMs baseada na redução do número de mensagens pela adoção de um determinado modelo de consistência de memória. Segundo esta classificação, os DSMs apresentaram duas grandes gerações:

1a. geração, caracterizada por um grande número de mensagens para manter a coerência;

2a. geração, caracterizada pela adoção de modelos de memória de liberação e seus derivados (de liberação preguiçosa, de entrada e de escopo), com redução drástica do número de mensagens.

Speight[72] propôs a seguinte classificação em três fases:

1a. geração onde é empregado o modelo de consistência seqűencial em em um ambiente de estações monoprocessadas; pode-se citar o Ivy[2] como exemplo;

2a. geração onde é empregado o modelo de consistência de liberação e derivados (por exemplo: Munin[10], TreadMarks[32]);

3a. geração caracterizada pela utilização de modelos de consistência de liberação e o *multi-threading* em um *cluster* com nós multiprocessados (Brazos[72]).

3.1 Proposta de uma nova classificação

Crê-se que a classificação de Carter é mais adequada do que a de Speight porque envolve somente os modelos de consistência e, pela sua adoção, a redução do número de mensagens de consistência pertinente. Portanto, foi proposta nos trabalhos[41, 42], uma extensão da classificação de Carter[10], mais adequada porque engloba modelos de consistência mais modernos. A classificação proposta é a seguinte:

1a. geração caracterizada por um grande número de mensagens para manter a coerência; um exemplo é o Ivy[37]

2a. geração caracterizada pela adoção de modelos de memória derivados do modelo de liberação (*release*) e redução do falso compartilhamento; os exemplos são o Munin[10] e o Quarks[10, 76];

3a. geração com modelos de consistência mais eficientes como o de liberação preguiçosa, de entrada e de escopo; os exemplos são o TreadMarks[32], o CVM[33], o JIAJIA[20] e o Nautilus[41].

Segue-se uma análise dos principais sistemas DSM.

3.2 Principais Sistemas Existentes

Esta seção analisa os principais DSMs existentes.

3.2.1 Ivy

Foi o primeiro sistema DSM proposto por Li[37] em 1986. A granularidade de acesso é uma página, e o protocolo de consistência é o *write-invalidate*, onde todos os nós que apresentam cópias não válidas de páginas recebem mensagens de invalidação das mesmas. A implementação deste protocolo pode se feita com um administrador central (que indica a localização de uma página), com vários administradores fixos (que vão administrar alguns conjuntos de páginas) e dinâmica (o nó proprietário da página contém a *copyset*, lista de todos os nós que possuem uma copia da página, podendo enviar as mensagens de invalidação da mesma)[48].

O protocolo de consistência é implementado por um administrador de páginas, que mapeia a memória física e administra a compartilhada. O modelo de consistência adotado é o seqűencial, onde a memória física age como *cache* do DSM.

3.2.2 Munin

O Munin[10, 11] foi o primeiro sistema DSM integrado à memória virtual e implementado em *software* que conseguiu uma redução drástica do número de mensagens pela utilização do modelo de consistência *release*.

Além disso, foi o primeiro DSM que introduziu a técnica de múltiplos protocolos de escrita para reduzir o efeito do falso compartilhamento[10, 11].

Com relação ao desempenho, foi um dos primeiros DSMs que conseguiu, para alguns aplicativos, desempenho próximo ao dos mesmos aplicativos convertidos para passagem de mensagens.

O Munin permite ao usuário a escolha de vários protocolos de consistência, como *write-invalidate*, *write-update* e *migratory*, podendo ser verificado qual deles é o mais adequado para o aplicativo que está sendo executado.

Para atacar o problema do envio de mensagens de atualização e invalidação que não são utilizadas pelo nó, porém enviadas já que estão na *copyset* do nó proprietário das páginas, o Munin desenvolveu o mecanismo de ***update timeout***, cessando-se o envio de mensagens de atualização

ou invalidação de páginas para nós que não forem mais usar as páginas (que estão envolvidas nas mensagens de atualização e invalidação)[48].

Para auxiliar a manutenção e organização das variáveis compartilhadas o Munin utiliza um diretório de objetos, que contém informações relevantes como os estados das páginas, *copyset*, provável proprietário, e outras. Conforme descrito anteriormente, a *copyset* é uma lista com todos os nós que se acredita ter uma cópia de uma página. Um nó é adiconado à *copyset* da página quando ocorre um pedido pela mesma. Quando um nó recebe a página pedida, recebe também uma cópia da *copyset* também. A tabela de estados de transição das páginas é análoga à da figura 2.4. O campo de provável proprietário é utilizado para encontrar um nó que certamente possua uma cópia da página. Em cada nó, existe um administrador de falta de páginas responsável pela implementação da tabela de transição de estados das páginas.

O Munin suporta semáforos e barreiras distribuídas, considerados na consistência *release* como pontos fundamentais para a manutenção da consistência.

Outro mecanismo importante que o Munin utiliza é a DUQ (*delayed update queue*): no início de uma seção crítica todas as váriáveis compartilhadas estão no estado de "somente leitura"; quando se tenta modificar qualquer uma delas, ocorre uma falta de página e o DSM faz uma cópia da página (*twin*), colocando-a na DUQ, e logo em seguida a passa para o estado de "escrita-leitura". Se a página já está no nó local, somente é considerado o *overhead* para a mesma trocar de estado. No *release*, é criada uma codificação palavra por palavra das modificações sofridas durante a seção crítica, isto é, os *diffs*, como pode ser visto na figura 2.5, sendo os mesmos colocados na DUQ. Ainda no *release*, são enviadas mensagens de contendo os *diffs* para os nós que possuem cópias das páginas, a fim de mantê-los coerentes.

3.2.3 TreadMarks

O TreadMarks é um dos sistemas DSM mais importantes. Foi o primeiro DSM a ter *speedup* comparável a uma máquina de memória fisicamente compartilhada para o *benchmark* SOR (da Universidade de Rice): basicamente comparando uma rede de máquinas DEC executando o TreadMarks com uma máquina de memória fisicamente compartilhada da SGI (440VGX)[32].

O modelo de consistência utilizado pelo TreadMarks é o modelo de consistência de liberação preguiçosa[32]. Segundo este modelo, a propagação das modificações sofridas durante uma seção crítica é atrasada até a próxima operação de *acquire* de semáforo.

Abaixo, tem-se o sumário das principais características do TreadMarks:

- modelo de consistência de liberação preguiçosa e suas variações a fim de minimizar o número de mensagens na rede;
- técnica de múltiplas escritas do Munin[10, 11];
- protocolos de consistência de invalidação, atualização e híbrido;

- primitivas compatíveis com o m4[1];
- implementado em forma de biblioteca;
- pode ser executado em rede de IBM SP2, de Suns Sparc e de PCs;
- pode ser executado em AIX, Solaris, FreeBSD e Linux 2.x;
- utiliza protocolo UDP para minimizar os *overheads* dos protocolos de rede.

Os *speedups*[38] do TreadMarks o tornam o principal DSM utilizado na comunidade acadêmica como nível de referência de *speedups* ótimos. Portanto, numa comparação entre DSMs, é essencial ter o TreadMarks para efeito comparativo. Pela utilização de múltiplos protocolos de escritas e do modelo de consistência de liberação preguiçosa, os *speedups* do TreadMarks[32] são bastante conhecidos na comunidade científica, tornando-o um dos sistemas DSM mais utilizados.

A eficiência do TreadMarks é derivada principalmente do seu modelo de consistência, o de liberação preguiçosa. Mas,o principal problema deste modelo é a alta necessidade de memória necessária para armazenar os *diffs* durante todas as seções críticas da execução do *benchmark*. Para avaliar o TreadMarks deve-se ter parâmetros de entrada que sejam um compromisso entre a memória livre do sistema, para que o sistema operacional não faça *swapping*, e suficiente para avaliar o *benchmark*, que é um compromisso entre computação e sincronização.

3.2.4 Midway

O Midway[6] utiliza o modelo de consistência de entrada para a manuteção da consistência.

Buscando as atualizações dos dados para manter a consistência ao se fazer o *acquire* do semáforo, o Midway diminui bastante a quantidade de comunicação utilizada para manter a consistência, bem como no *release*, e ao contrário da consistência tipo *release*, nenhuma informação de consistência precisa ser propagada ou transmitida.

Abaixo tem-se o sumário das principais características do Midway:

- consistência de entrada para minimizar o número de mensagens pela rede;
- detecção das modificações das variáveis por meio de códigos introduzidos por um compilador, desta forma não utiliza a técnica de múltiplas escritas concorrentes do Munin;
- existe um administrador de barreiras que recebendo as mensagens de atualização vindas de outros nós, as combina, reenviando estas últimas (combinadas) para os outros nós da rede.
- utiliza protocolo de atualização.

[1] m4: padrão contendo primitivas para sistemas de memória compartilhada.

A característica mais interessante do Midway é o modelo de consistência de entrada, por ele adotado. O grande inconveniente deste modelo justamente é a associação de dados compartihados a variáveis de sincronização e, deste modo, acaba exigindo do usuário a mudança na sua forma de programar, o que pode acabar sendo bastante complicado[48]. Outro inconveniente é a dependência do compilador para a inserção dos códigos necessários para detectar as modificações sofridas durante uma seção crítica. O compilador gcc estava sendo modificado para isso em 1993, porém este projeto aparentemente não foi concluído.

3.2.5 Quarks

O Quarks[9] é um importante DSM porque foi um dos primeiros DSMs de domínio público a serem implementados. Também foi um dos primeiros sistemas DSM de domínio público a utilizar a consistência de liberação para a minimização do número de mensagens de consistência através da manutenção da mesma somente nos pontos de sincronização, *release* e *acquire*. De acordo com este modelo, *diffs* são transmitidos no momento do *release* para manter a consistência. Abaixo tem-se as principais características do Quarks:

- utiliza a consistência de liberação;
- a técnica de múltiplos protocolos de escrita do do Munin[10, 11];
- utiliza replicação de páginas e protocolos de atualização e invalidação;
- as primeiras versões do Quarks utilizavam o pacote *pthreads* para diminuir o chaveamento de contexto; porém segundo o estudo de Carter[76], não faz nenhuma diferença o fato de ser *multithreaded*;
- primitivas compatíveis com o TreadMarks, JIAJIA;
- utiliza protocolo UDP e *sockets*;
- rede de Suns Sparc e PCs;
- implementado em forma de biblioteca.

Apesar de ter características bastante interessantes, como ser de domínio público e de apresentar o modelo de consistência de liberação, que na época quando foi lançado (1995), não era o de maior desempenho, o Quarks foi um dos pioneiros para serem executados em redes de PCs. Na realidade, as características do Quarks são muito semelhantes às do Munin, já que seu criador foi a mesma pessoa: Carter.

O grande fator limitante do Quarks é o modelo de consistência *release*. Desde então, foram propostas algumas modificações[76] em 1998, através de simulações, mostrando que ainda poderia

ser um DSM que poderia competir com os demais. Porém, estas modificações só ficaram nas simulações, já que até a apresentação do trabalho de 1998, nenhuma nova versão surgiu desde 1996.

3.2.6 CVM

O CVM [33, 35] é um DSM de domínio público, projetado por Keleher. Da mesma maneira que o TreadMarks[32], assunto principal da tese de Keleher[32], o CVM apresenta características bastantes semelhantes, que começam na adoção do mesmo modelo de consistência: o de liberação preguiçosa. Segundo este modelo, a propagação das modificações sofridas durante uma seção crítica é atrasada até a próxima operação de *acquire* de semáforo.

Seguem-se as principais características do CVM[33]:

- modelo de consistência de liberação preguiçosa[32];
- implementado com objetos;
- *multithreaded*, para minimização do chaveamento de contexto;
- apresenta versões para Solaris e AIX, mas até a data de apresentação deste trabalho não apresenta versão para Linux 2.x.

Em [10, 41], mostra-se que pelo fato do CVM adotar o modelo de consistência de liberação preguiçosa, apresenta problemas parecidos com o TreadMarks: grande quantidade de memória necessária para armazenar os *diffs* durante todas as seções críticas da execução de *benchmarks*, o que pode acarretar o *swapping* do sistema operacional em certas condições.

3.2.7 Brazos

O DSM Brazos[72, 73] é o único DSM a utilizar o *multicast*, uma implementação em *software* da consistência de escopo e vários mecanismos adaptativos de *runtime* para aperfeiçoar seu desempenho. O sistema de *runtime* do Brazos é *multithreaded* permitindo a sobreposição de computação com o *overhead* de comunicação tipicamente associado com *softwares* DSMs. O Brazos permite que os programas aplicativos aproveitem os vários processadores de uma máquina de memória compartilhada física pelo uso do *multithreading*, enquanto transparentemente interage com o sistema de memória virtual compartilhada.

O Brazos utiliza o *multicast* seletivo para reduzir o número de mensagens de consistência e implementar eficientemente o modelo de escopo. Utiliza a implementação em *software* do modelo de escopo para reduzir o número de mensagens de consistência e o efeito do falso compartilhamento. Incorpora também o uso de mecanismos que amenizam o efeito do *multicast* pela redução das *copysets* assim como o mecanismo que melhora o *throughput* da rede.

O Brazos foi desenvolvido para tirar vantagens das características do Windows NT, inclusive o *multithreading*[52, 80] preemptivo, suporte de *multicast* no TCP/IP, suporte multiprocessado do NT. O sistema Brazos em si é *multithreaded*, permitindo que longas latências de comunicação associadas com DSMs sejam superpostas com processamento. Permite também que programas tirem vantagem do multiprocessamento local. A coerência entre os *threads* do Brazos em nós SMP é mantida pelo *hardware*.

Existem dois *threads* principais no Brazos que são responsáveis por responder rapidamente aos outros processos e a pedidos previamente enviados pelo processo. O uso de um *thread* separado para atender múltiplos pedidos permite ao Brazos manter vários pedidos ao mesmo tempo, o que pode melhorar significativamente o desempenho.

Para reduzir o número de mensagens de consistência e implementar de maneira eficiente a consistência de escopo, o Brazos utiliza as primitivas de *multicast* da biblioteca Winsock 2.0. Em um ambiente que é multiplexado ao longo do tempo, enviar uma mensagem de *multicast* não é mais caro do que enviar uma mensagem ponto a ponto, reduzindo-se tanto o número de mensagens como a quantidade de bytes transferidos para manter a consistência.

Quando um processo chega a uma barreira, o mesmo envia uma mensagem a um administrador estático de barreiras indicando que o mesmo chegou a este ponto de sincronização. Quando o administrador de barreira recebe a notificação que todos os processos chegaram (a barreira), ele recebe nessas mesmas mensagens as páginas que foram invalidadas pelos mesmos. Então, o administrador envia para cada processo uma mensagem com todas as páginas a serem invalidadas, o nó que as invalidou e o processo está livre para sair da barreira. Os *threads* então irão sofrer as faltas das páginas que foram recebidas na barreira. Para obter os *diffs* de modo a atualizar as páginas, o administrador de cada nó envia uma mensagem de *multicast* a todos os processos que tem cópias da página. Esses processos irão responder com uma mensagem de *multicast* não somente para o processo que pediu, mas também para para os processos que estão na *copyset* (lista de nós que o processo corrente acredita ter uma cópia da página). Estas mensagens de *multicast* irão conter *diffs*, codificações das mudanças sofridas pela página durante a última invalidação. Processos que precisam dos *diffs* e que ainda não sofreram falta de páginas irão receber *diffs* indiretos por essas páginas, com a intenção de trazer a página atualizada antes a falta de página resultante da ocorrência de página inválida.

Na figura 3.1, os processos P0, P1 e P2 cada um escrevem (W(0), W(1) e W(2)) em uma mesma página compartilhada algum tempo antes da primeira barreira barreira. Após a primeira barreira cada processo lê um valor escrito or outro processo indicado por R(0), R(1) e R(2). Os pedidos são mostrados por flechas tracejadas e as respostas por linhas contíguas. No total, 12 mensagens ponto a ponto são utilizadas para atualizar os processos P0, P1 e P2, e somente 3 quando se utiliza o *multicast*. O processo P3 também recebe *diffs* indiretos da página, embora eles não sejam utilizados.

A outra grande vantagem é a economia de *diffs* pelo uso do *multicast*, pois em outros DSMs como o TreadMarks, é necessário manter uma lista dos *diffs* passados até que os outros processos

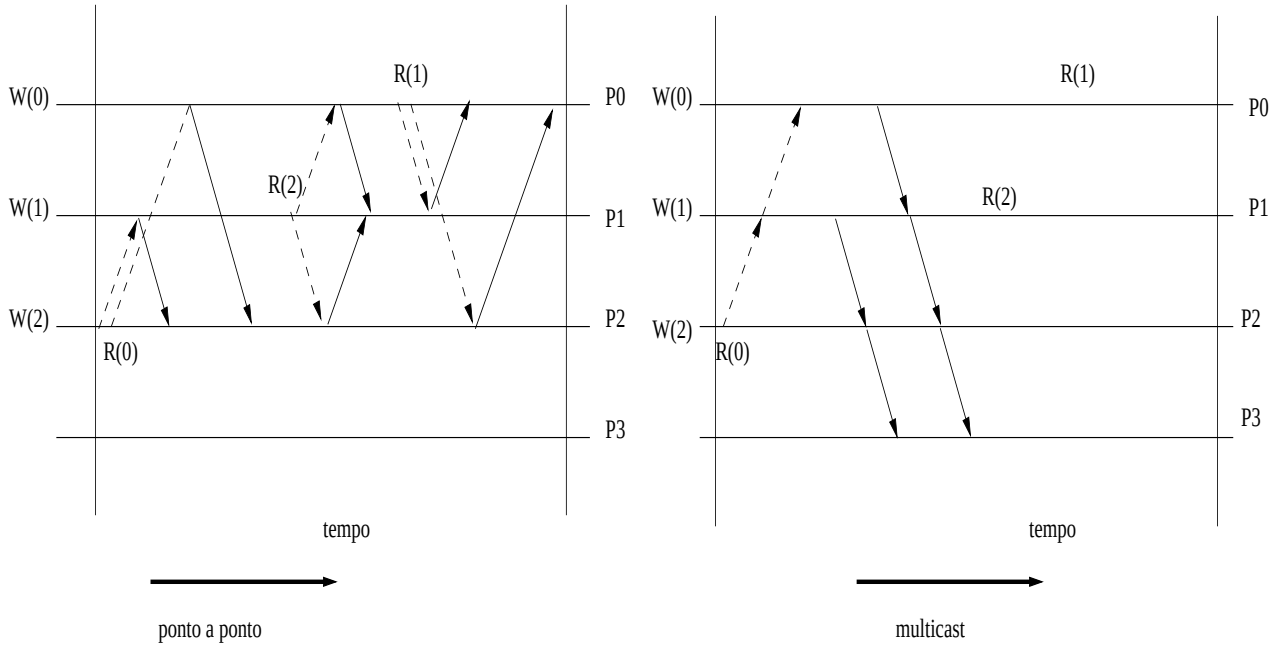

Figura 3.1: Comunicação ponto a ponto versus *multicast*[72, 73]

tenham visto os *diffs*[72]. Isto acaba gerando uma grande utilização de memória para guardá-los e o processo de *garbage collection* para descartar os que não mais são necessários. No Brazos, pelo uso do *multicast*, todos os processos que estão na *copyset* da página recebem os *diffs* indiretamente, portanto somente é necessário que se guarde um *diff* por página, ao invés de vários[72].

Problemas com o *multicast*[73]:

- o *multicast* é dependente do hardware;
- para redes de PCs, o *multicast* não é muito escalável, podendo ser empregados de 8 a 16 nós;
- exige uma rede isolada;
- nós que não utilizam as páginas não necessitam receber mensagens de *multicast*; o Brazos utiliza um algoritmo para reduzir a *copyset*;
- conflitos de *multicast*: quando um processo recebe um *diff* indiretamente enquanto um *thread* ou processo está esperando por *diffs* para serem aplicados à mesma página; este problema acaba causando um tráfego de rede bem maior, apesar do processo de atualização antecipada do Brazos reduzir estes conflitos.

No estudo de Speight[72], são comparados uma versão *single-threaded* do TreadMarks, uma versão *multithreaded* do TreadMarks e o Brazos (que incorpora o *multithreading* e o *multicast*), sob seis programas científicos. O uso do *multicast* e *multithreading* comparados ao *multithreading* resultaram em um aumento de desempenho de 38%. Sobre o *multithreading*, 5 das 6 aplicações estudadas se beneficiaram em cerca de 17%.

3.2.8 JIAJIA

Como o JIAJIA é um dos DSM de importância fundamental para este trabalho, será descrito mais detalhadamente.

Funcionamento Geral do JIAJIA Ao contrário de outros *softwares* DSM que adotam o esquema COMA (*cache only memory access*), o JIAJIA organiza a memória compartilhada de acordo com o esquema NUMA(*non uniform memory access*)[22]. Segundo este mecanismo, cada página compartilhada apresenta um nó fixo e os *homes* das páginas são distribuídos ao longo de vários nós. Quando *cached* as páginas remotas são mantidas no mesmo espaço de endereço de usuário assim como no nó *home*. Desta forma o espaço de endereçamento compartilhado de uma página é idêntico em todos os processadores e nenhuma translação de endereços é necessária em um acesso remoto.

No JIAJIA páginas compartilhadas são alocadas com a primitiva *mmap()*. Segundo Shi[22], cada página compartilhada apresenta um endereço global fixo que determina o *home* da página, sendo que inicialmente a página é mapeada somente no espaço de endereçamento do seu nó *home*. Uma referência a uma página não *home* causa um sinal SIGSEGV. O *handler* para o SIGSEGV então mapeia a página que sofreu a falta, do espaço de endereçamento global para o espaço de endereçamento local. Como a memória local alocada pode ser maior que a memória física de um só nó, o mapeamento de muitas páginas remotas poderá derrubar o sistema. Para evitar isto, cada nó mantém um *cache* para armazenar as páginas cujo *home* não é o próprio nó. Se o número de páginas remotas localmente é maior que o número máximo de páginas permitido, é retirado o mapeamento de algumas páginas mais antigas para dar lugar a uma nova página.

Com a organização de memória acima proposta por Hu[22], o JIAJIA é capaz de suportar memória compartilhada maior que a memória física. Em outros DSMs como o TreadMarks, CVM e Quarks, o espaço de endereçamento compartilhado é limitado pela memória física porque nenhum mecanismo de *replacement* de *caches* é implementado e portanto cada nó deve suportar todas as páginas compartilhadas. Além disso, neste sistema cada processador mantém uma tabela de páginas local para manter o diretório de páginas, *twins*, *diffs*, estados das páginas, endereço local e global da página. O tamanho desta tabela aumenta linearmente com o número de páginas compartilhadas. No JIAJIA, os *homes* das páginas compartilhadas são distribuídos e a tabela de página somente contém informação das páginas *cached*, mantendo somente a informação de seu endereço, estado e dos *twins*.

Outra característica importante que o JIAJIA permite é a flexibilidade de o programador controlar a distribuição inicial dos *homes* dos dados compartilhados. Existe uma função que permite ao programador alocar certos tamanhos de memória compartilhada a cada nó[22]. O processador onde a alocação começa pode ser indicado.

O JIAJIA utiliza o modelo de consistência de escopo. Adotando o modelo de escopo, simplifica-se bastante o protocolo *lock-based* de coerência de *cache*. No TreadMarks, estruturas complexas

como *intervals*, *time-stamps*, sendo empregados para implementar uma relação *happen-beffore-1[Adve93] apud* [32] dos acessos à memória. No JIAJIA, somente os intervalos prévios relacionados ao *lock* a ser adquirido devem ser visíveis ao processador que está fazendo o *acquire*. A coerência é mantida pela escrita e leitura de *write-notices* no *lock*[22].

No JIAJIA, quando ocorre um *release,* o nó que o está executando executa uma comparação de todas as páginas escritas em sua seção crítica obtendo-se os *twins* e os *diffs* a eles associados. Depois de todos os *diffs* serem aplicados, uma mensagem de *release* é enviada ao *home* do *lock* associado, para que ele seja liberado. Além disso, o nó que está fazendo o *release* junta, na mensagem de *release*, os *write-notices* daquela seção cítica, notificando as modificações ocorridas durante a mesma[22].

Por outro lado, no *acquire* o processador que o está executando envia o pedido de um *lock* ao administrador. O processo que pede fica parado até que ele consiga o *lock*. O administrador ao enviar a permissão para o nó fazer o *lock,* envia junto com esta, as *write-notices* associadas a ele. Depois do processador que está fazendo o *acquire* receber a mensagem de autorização do administrador, ele invalida todas as páginas *cache* que são notificadas como obsoletas pelas *write-notices* associadas[22].

Uma barreira pode ser vista como um par *unlock-lock*. A chegada à barreira constitui o término de uma seção crítica e, a saída constitui o início de uma nova. Duas barreiras envolvem uma seção crítica. Numa barreira, todas as *write-notices* associadas a todos os *locks* são "*reseted*"[22].

Com relação às falhas de página no JIAJIA, numa falha por leitura, a página é buscada do nó *home* para a memória local. Numa falha por escrita, se a página não está presente, ou se está no estado INV, é buscada do *home* no estado de RW. Se a página escrita está no estado de RO, o estado é modificado para RW. Uma *write-notice* relativa a esta página é armazenada e um *twin* é criado antes de ser escrita[22].

Como se pode perceber, todas as ações relativas à coerência são feitas nos instantes de sincronização, portanto menos *oveherads* relacionados a escritas e leituras ocorrem. Além disso, o *overhead* para manter o diretório é menor[22].

3.2.8.1 Técnica Adaptativa de Detecção de Escrita[55]

O trabalho de Hu[23], compara o desempenho do algoritmo de detecção tradicional de falta de páginas detectadas pela memória virtual com o onde se protege tanto o nó *home* e os nós que têm *caches* das páginas no começo de um intervalo, dando permissão de escrita somente nos *caches* e não nos nós *home*, permitindo que não detecte as escritas nos nós *home* e invalide os *caches* no começo no intervalo. Mostra também uma API para detectar as escritas que requer que o programador ou pré-compilador explicite as escritas no programa.

Ainda no trabalho de Hu[23], é proposto o esquema CO-WD ou *cache only write-detection* para o DSM JIAJIA, onde não se detectam as escritas nos nós *home*. Somente as escritas nas cópias *cache* é que são detectadas. As escritas nos *caches* são detectadas e geram *diffs* que são enviados

para os nós *home*. Este esquema invalida todas as cópias *cache* no começo de um intervalo. Uma API (API-WD) requer que o programador ou compilador explicite os momentos onde as escritas vão ocorrer no programa.

O esquema CO-WD[23] detecta as escritas através de faltas de página. Estudos com o JIAJIA mostram que proteger contra escrita as páginas nos nós *home* causam *overheads* significantes em aplicações com grandes conjuntos de dados compartilhados, onde a maior parte das escritas se faz no nó *home*. Este esquema reduz o *overhead* de detecção de falta de páginas ao custo de alguns *misses* extras. Neste esquema todas as páginas *cache* são assumidas serem obsoletas e invalidadas no começo do intervalo. Nos nós *home* as escritas não são detectadas porque o propósito de detectar as *write-notices* das páginas *home* é manter a coerência através de invalidações associadas às páginas *cached* e o esquema CO-WD invalida todas as páginas *cached*, quando o intervalo começa. Somente as escritas a páginas *cached* são detectadas para gerar *diffs* que são enviados aos nós *home* no final do intervalo.

Detalhando mais o esquema CO-WD[18], quando um processo tenta escrever uma página compartilhada num intervalo, um sinal de SIGSEGV é disparado pelo SO e a *write-notice* da página é criada. Então existe um *handler* apropriado que trata este sinal e permite agora que a página seja escrita. No fim do intervalo, as *write-notices* associadas às páginas são enviadas junto com o *lock* no *release* do mesmo. Se a página *home* for escrita no próximo intervalo e mantida em estado *writable* ao longo do mesmo, economizam-se vários *mprotects()* e SIGSEGVs. Se um nó escrever em sua cópia *cached* da página, o nó *home* volta a ter proteção contra a escrita no próximo intervalo e todas as outras cópias exceto o *home* são invalidadas. Este raciocínio se baseia no comportamento *single-writer* das aplicações. Esta técnica adaptativa então reconhece uma única escrita a uma página compartilhada pelo seu *home* e assume que a página fique *writable* no *home* até que no futuro a página seja escrita por algum nó que possua uma cópia *cached* da página.

No esquema da API-WD [23], cada nó contém um bit para cada página que é *home* e um vetor de bits para cada página *cached*. Zerando o bit da página que é *home,* significa que a página foi alterada, enquanto que zerando o i-ésimo bit do vetor indica que a i-ésima palavra da página é modificada. Existe uma função que guarda as escritas às variáveis compartilhadas em uma certa região. O programador deve se responsabilizar por inserir uma instrução desse tipo a cada escrita de variável. Em cada começo de intervalo o *dirty* bit de cada página *home* e o vetor de bits são resetados. No final de cada intervalo, estes bits são checados para averiguar qual das páginas *cached* foram modificadas.

3.2.8.2 Redução dos *overheads* de detecção de escrita[22]

Os múltiplos protocolos de escrita necessitam detectar escritas feitas na memória compartilhada de forma que o protocolo possa ser ativado para corretamente a propagação das escritas. Dois metodos são empregados: o tradicional VM-WD (*virtual memory write detection*), e o CO-WD (descrito acima), sendo que este último não protege contra escrita as páginas *home*, mas invalida

todas as páginas *cached* no começo do intervalo.

No esquema VM-WD, bastante tradicional nos DSMs, tanto as páginas *home* e *cached* são inicialmente protegidas contra a leitura no começo de um intervalo. Para uma uma escrita em uma página *cached*, um *twin* da página é criado e uma *write-notice* da página é armazenada no *handler* de SIGSEGV. A proteção de escrita é eliminada até o final da seção crítica onde as escritas na página ocorrem sem faltas de página. No final do intervalo, uma comparação palavra por palavra é feita entre a página escrita e a *twin* para produzir o respectivo *diff*. As *write-notices* e os *diffs* são enviados respectivamente para o administrador do *lock* e para os *homes*. Se o SIGSEGV é causado por um falta por escrita em um *home*, então a proteção contra escrita é eliminada. No final do intervalo, escritas sobre os nós *home* são enviadas pelo *lock* associado.

O esquema VM-WD detecta escrita através de faltas de página. Para aplicações com grande quantidade de dados compartilhados e boa distribuição de dados, o esquema CO-WD reduz o *overhead* de detecção de páginas *home* ao custo de algumas faltas de página extras, quando necessário. Resumindo [18, 22], neste esquema todas as páginas *cached* são assumidas obsoletas e invalidadas no começo do intervalo.

3.2.8.3 Redução dos *overheads* de falso compartilhamento[22]

Se a modificação de uma página é feita pelo processador e a página não for modificada por nenhum outro nó então a invalidação é desnecessária, mantendo assim o acesso correto a páginas por ele modificadas anteriormente e por nenhum outro nó.

Se a página for modificada somente nos outros nós (não *home*) e não existir nenhum outro nó que a modifique desde a última barreira, é desnecessário enviar o *write-notice* associado àquela barreira ao administrador. Para isto, um *read-notice* é armazenado sempre quando o *home* de uma página é pedida de algum outro nó da rede.

O método *incarnation number* descrito por Bershad[6] e por Iftode[23, 26], elimina invalidações desnecessárias. Cada *lock* é associado com um número, que é incrementado sempre que um *lock* é transferido. Além disso, cada processador mantém um *incarnation number* para cada *lock*. Quando uma *write-notice* é armazenada no *lock*, o *incarnation number* também é armazenado. Num *acquire*, o processador inclui o *incarnation number* do *lock*. Na liberação do *lock*, o número do *lock* e os *write-notices* que apresentaram *incarnation number* maior do que o nó pedido são enviados para o processador que faz o *acquire*. O processador que faz o *acquire* então seta seu *incarnation number* local do *lock* para o do recebido pela aquisição do *lock* e invalida as páginas *cached* de acordo.

3.2.8.4 Propagação Hierárquica de Mensagens em Barreiras[22]

As barreiras ainda são uma forma cara de sincronização entre processadores. Existe normalmente um administrador para cuidar dos pedidos dos processadores, que no JIAJIA é responsável por

combinar as *write-notices* recebidas de todos os processadores e enviá-las para cada um. O processamento nas barreiras de forma seqűencial produz um gargalo potencial no sistema quando o número de processadores é grande.

O JIAJIA provê uma estrutura em forma de árvore para minimizar este problema. Mapeia todos os nós em uma árvore binária e o nó raiz é o administrador. Quando um processador chega a uma barreira ele não envia pedidos de barreira ao administrador. Ao invés disto, envia pedidos aos pais na árvore. Depois de receber os pedidos de ambos os filhos, os pais combinam os *write-notices* e enviam o pedido aos pais. Todos os processadores chegam à barreira quando o nó raiz receber pedidos de ambos os filhos. Da mesma forma com relação aos *acks,* são propagados da raiz para cada nó de maneira inversa.

3.2.8.5 *Home-based*

O trabalho de Iftode[24] mostra que os softwares DSMs *home-based* são escaláveis, simples e de bom desempenho. Num protocolo *home-based*, cada página compartilhada apresenta um nó *home*, para onde são propagadas as escritas e de onde todas as cópias são derivadas. Num *multiple writer protocol*, os *diffs* são enviados para os *homes* associados no fim dos intervalos e páginas *cached* são invalidadas. Sobre o protocolo LRC, os *home-based* apresentam as seguintes vantagens: cada falta de página requer somente uma *round trip* completa ao *home*; escritas nos nós *homes* não produzem *twins* e *diffs*; *diffs* são enviados para os *homes* no próximo instante de sincronização consumindo menos memória; uma cópia da página pode ser trocada com o *home* quando a memória local excede o número de páginas do *cache*, enquanto em softwares DSMs que seguem o modelo COMA, deve existir um mecanismo que garanta que a última cópia compartilhada não seja trocada; estados de coerência mais simples.

O desempenho dos DSMs *home-based* é sensível a distribuição de dados. Se os *homes* forem bem distrubuídos, os acessos em cada nó caem localmente, acarretando um bom desempenho. A produção e aplicação de *diffs* antecipadamente pode introduzir mais *overheads* do que produzir e aplicar *diffs* atrasadamente como no modelo LRC. No modelo *home-based*, se o *home* for o produtor do *diff*, o esquema *home-based* é superior pois nenhuma produção ou aplicação de *diffs* é necessária. O produtor escreve diretamente no próprio nó e o consumidor o aplica pela busca da página. Se o *home* da página estiver no consumidor, o produtor produz os *diffs* no *release* e os envia ao consumidor (*home*), sendo o acesso aos dados feito através do *home*. Se o *home* estiver num terceiro nó, o mecanismo *home-based* requer mais mensagens que o não *home-based* porque o produtor e consumidor devem se comunicar com o *home*.

3.2.8.6 *Home-migration*

Um bom esquema de distribuição de páginas *home* deve manter o *home* de cada página mais próximo do nó que mais freqűentemente escreve na mesma. O esquema mais simples de resolver é fazer o *broadcast* com o novo *home* de todas as páginas migradas. Para reduzir o *overhead* do

broadcast, a decisão da migração da página é feita no administrador de barreira e a migração é feita junto com as mensagens de *release* da barreira. Portanto, no esquema de migração proposto por Hu[21] para o JIAJIA[20], páginas escritas por um único nó são migradas para o nó onde são escritas. As mensagens de migração são agregadas às mensagens das barreiras, não requerendo nenhuma mensagem adicional.

Explicando melhor o esquema de migração proposto por Hu[21], o JIAJIA migra o *home* da página para o novo nó somente se o novo *home* for o único nó escritor da página durante o último intervalo. No protocolo básico, uma página *cached* é invalidada na barreira se a página foi modificada no intervalo da última barreira. Portanto se a página é modificada por um único processador durante a última barreira e o processador que modificou a página não é o *home*, então não é necessário fazer com que este processador invalide a página porque esta já está coerente, e as outras cópias são invalidadas normalmente. Resumindo-se o esquema de Hu[21]: na chegada de uma barreira, o processador que chega envia *write-notices* do intervalo de barreira para o administrador. Se a página associada é válida no *cache* um "1" vai estar na mensagem. Na recepção de um pedido de barreira, o administrador armazena os *write-notices* embutidos no pedido. Se mais de um nó modificou a mesma página, um *tag* é setado como MULTIPLE. Depois de todos os pedidos de chegada de barreira terem sido recebidos, o administrador de barreira faz um *broadcast* com todas *write notices* e os tags respectivos para os nós. Na recepção da saída da barreira, cada processador checa seu *write-notice* e o *tag*. Se a *write notice* foi modificada por vários nós e se o nó que modificou não é o próprio nó e nem o *home*, então a página é invalidada. Se a *write-notice* e o modificador indicar que a página foi modificada por um só nó e o nó tem uma cópia válida ("1" setado), então o *home* é migrado para o nó. Se o nó é migrado, neste nó o *cache* deve ser liberado e o novo *home* configurado como sendo este nó. Se o *host* é o nó *home* antigo da página migrada, o item associado no *home* é liberado, e a página é invalidada, bem como o *home* é configurado como o novo nó. Se o *home* não for nem o novo *home*, nem o antigo, somente o item no campo de *home* para apontar para o novo *home* é atualizado.

O estudo de Hu[20] conclui que a utilização de *home-migration* acaba gerando mais faltas de página.

3.2.8.7 Implementação interna da tabela de páginas

No JIAJIA a tabela de páginas precisa somente de seis bytes para cada página, incluindo um inteiro para indicar o *home* da página, um índice para o *array home* e um índice para o *array cache*. Cada item no *home array* inclui o endereço global da página, um *tag* para indicar se a página foi modificada e um *tag* para indicar se a página está *cached* em outros nós. Cada item do *cache*, inclui endereço, estado da página, ponteiro para o *twin* da página.

3.2.8.8 Diagrama de estados da consistência *scope* com o protocolo *lock-based*

O diagrama de estados das páginas do DSM JIAJIA pode ser acompanhado pela figura 2.8. Num *release*, o nó que o executa calcula os *diffs* pela comparação dos *twins* com as páginas. Os *diffs* são enviados para os *homes*. Depois de todos os *diffs* serem aplicados nos *homes*, uma mensagem é enviada para o administrador do *lock* para liberar o mesmo. As indicações das *write-notices* são adicionadas às mensagens de *release* indicando quais as páginas modificadas na seção crítica.

No *acquire*, o processo que o executa envia um pedido pelo *lock* no administrador e fica *stalled* enquanto não recebe autorização para utilizá-lo. Ao permitir o uso do *lock*, o administrador coloca as *write-notices* associadas com o *lock* na mensagem de liberação para que o nó que requisitou o *lock* possa utilizá-lo. Depois que o nó que fez o pedido de *acquire* recebe a notificação para utilizá-lo, ele invalida todas as páginas indicadas nas *write-notices*. Todas as *write-notices* de todos os *locks* são resetadas nas barreiras. Numa falta por leitura, a página é buscada do nó *home* e colocada no estado read-only (RO). Numa falta por escrita, a página é buscada e colocada no estado RW. Se uma página *cached* está no estado RO, passa ao estado RW e o respectiva *twin* para o cálculo do *diff* é criada.

3.2.8.9 *Overhead* de Comunicação

Em geral o *overhead* de comunicação e o *overhead* induzido devido à coerência são os principais responsáveis pela perda de *performance* em sistemas DSM. Baseado no trabalho de Shi[69] sobre o *software* JIAJIA, algumas conclusões foram tiradas:

1. tempo gasto na interrupção quando sobreposto com o tempo de espera (40.94%) em média;

2. codificação e decodificação de *diffs* não gasta muito tempo ($<$1%);

3. operações de sincronização levam 13.65% no tempo de execução;

4. o *overhead* de comunicação do *hardware* domina 2/3 do tempo total de comunicação e o *overhead* do *software* de comunicação não ocupa muito tempo. O principal gargalo é a banda e não a latência induzida pelo *overhead* do *software* de comunicação.

3.2.8.10 Comparação com outros DSMs

Em [20], onde o JIAJIA foi caracterizado e descrito, Hu fez uma comparação deste DSM com o CVM, apresentando desempenhos melhores em vários aplicativos. Quando o desempenho foi pior que o CVM, foi pouco pior(inferior a 5%).

3.2.8.11 Comunicação a nível de usuário

No estudo de Shi[67], são propostas primitivas de comunicação a nível de usuário novas que serão futuramente implementadas para o DSM JIAJIA. Além disto, no estudo anteriormente citado, são descritos os principais fatores de projeto a nível de sistema de comunicação que devem ser levados em conta quando se tem por meta obter bom desempenho de um sistema DSM.

3.2.8.12 Conclusão

Resumindo, seguem-se as principais características do JIAJIA:

- modelo de consistência de escopo baseado em *homes*, minimizando o número de mensagens na rede;
- técnica de múltiplas escritas concorrentes do Munin[10, 11];
- proprietários fixos das páginas;
- distribuição de dados possível de ser escolhida pelo usuário ao longo dos nós;
- implementado em forma de biblioteca;
- primitivas compatíveis com o TreadMarks;
- pode ser executado em rede de IBMs SP2, Suns Sparc e PCs;
- pode ser executado em AIX, Solaris, FreeBSD e Linux 2.x;
- utiliza protocolo UDP para minimizar os *overheads* dos protocolos de rede.

O principal objetivo do JIAJIA[69, 20, 16] é ser o mais simples possível, a fim de minimizar os *overheads* de criação e armazenamento de *diffs*, minimizando o número de mensagens emitidas para a rede.

A característica mais interessante do JIAJIA é sua simplicidade: baseado em *homes*, assim os *diffs* são somente transmitidos para os proprietários das páginas e não para vários nós, minimizando o número de mensagens na rede; o usuário conhecendo o comportamento do seu programa escolhe uma distribuição de dados que é mais apropriada, ou seja, que lhe proporcione melhores *speedups*.

3.2.9 ADSM

O ADSM [50] é um DSM baseado em consistência *lazy release* que constantemente se adapta aos padrões de compartilhamento das aplicações. Este DSM é implementado em cima do TreadMarks. Essa adaptação é baseada em uma categorização dinâmica do tipo de padrão de compartilhamento esperado de uma página. As páginas podem ser classificadas em falsamente compartilhadas, migratórias, ou tipo produtor/consumidor. As páginas que se encaixam no comportamento migratório e produtor/consumidor são administradas em modo *single writer* enquanto que as falsamente compartilhadas são administradas em modo *multiple writer*. A coerência é mantida por meio de invalidações, mas as atualizações são utilizadas por dados protegidos contra *locks* em comportamentos migratórios e dados protegidos contra barreiras em comportamento produtor/consumidor.

Os resultados de Monnerat[50] mostraram que o ADSM consegue superar o TreadMarks em até 155% e o TreadMarks adaptativo em 67%. Mas, curiosamente, o estudo mais recente de Amza[5], através de técnicas adaptativas aplicadas ao TreadMarks, consegue melhora substancial somente em alguns poucos programas, chaveando-se de modo *multiple writer* para *single writer*.

3.2.10 DASH [12] apud [32]

É um projeto de um multiprocessador coerente a *cache*, onde os processadores estão agrupados em *clusters*, e em cada *cluster*, compartilham o mesmo *bus*. Os vários *clusters* estão conectados a uma rede em *mesh*. Dentro do *cluster*, a coerência é do tipo *snoopy* e o protocolo de coerência é baseado em diretórios. O protocolo de consistência é baseado em invalidações e o modelo de consistência é do tipo *release* nas linhas do *cache*.

3.3 Outros sistemas

Alguns outros sistemas como o Adsmith [3], Cashemere[12], DIPC[15], Larchant[36] , Mirage [49], Peregrine[29], Proteus[81], Shasta[64], SHRIMP[70], Simple COMA[71], SVMlib[65], Vote[82] e Wind Tunnel[85] não foram detalhados porque não tem relevância fundamental para este trabalho.

Capítulo 4

Especificação do Sistema DSM Nautilus

Neste capítulo, são apresentadas e justificadas as principais características do sistema DSM Nautilus.

4.1 Proposta Geral

A principal motivação da criação do Nautilus é desenvolver um DSM que apresenta um modelo de consistência de memória eficiente, de implementação simples e que proporcione bons *speedups*[30]. O Nautilus apresenta uma interface simples com o usuário e também é compatível com o TreadMarks e JIAJIA.

O Nautilus segue a tendência internacional, sendo implementado em forma de biblioteca.

O Nautilus apresenta uma organização baseada em páginas. Segundo este esquema, as páginas são replicadas entre vários nós do *cluster*, permitindo múltiplas leituras e escritas, portanto permitindo bons *speedups*. Utiliza o mecanismo de múltiplas escritas concorrentes do Munin[10, 11], objetivando reduzir o falso compartilhamento.

O protocolo de consistência adotado pelo Nautilus é o de invalidação porque vários estudos [10, 11, 22, 32] mostraram que este tipo de mecanismo provê bons *speedups* para diferentes aplicações genéricas.

O modelo de consistência adotado é o de escopo, e que é implementado através do protocolo de coerência baseado em *locks*, com invalidações.

O Nautilus é diferente dos outros sistemas DSMs existentes em vários aspectos[39, 43, 44]. Primeiro, a sua implementação é *multithreaded*, (com processos leves utilizando o pacote *pthreads*) minimizando, portanto, o chaveamento de contexto. Além disso, também não utiliza o sinal SIGIO (para detecção da chegada de uma mensagem). O Nautilus administra a memória compartilhada utilizando o esquema *home-based*, como o JIAJIA faz, porém com uma estrutura de diretório que contém todas as páginas, ao invés de somente uma estrutura *cached*, isto é, com as páginas relevantes, como implementado no JIAJIA.

A implementação da administração de memória segue o protocolo de coerência *lock-based*.

Pelo fato de ser *multithreaded*, implementado em plataforma Linux, utilizar o modelo de consistência de escopo, o Nautilus é único com todas estas características. As características do Nautilus podem ser vistas no item 1.4.

4.2 Modelo de Consistência

Devido às vantagens do modelo de consistência de escopo e devido à simplicidade de sua implementação, o Nautilus adotou este modelo, que fornece bons *speedups*[35, 42].

Não foram adotados:

1. modelo de consistência de liberação preguiçosa porque este acaba utilizando uma grande quantidade de memória para armazenar todos os *diffs* de todas as seções críticas ou até o processo de coleta de lixo ser efetuado. Outro problema é o *overhead* para a obteção dos *diffs* de vários nós distintos.

2. modelo de consistência de entrada porque, para associar dados compartilhados a semáforos, o programador deve modificar seus programas, o que pode ser difícil, dependendo das estruturas de dados nele contidas, podendo comprometer bastante a compatibilidade.

3. modelo de consistência de liberação porque, apesar de fácil implementação, acaba gerando uma grande quantidade de mensagens[32] devido a replicação das páginas e o envio de mensagens de consistência para mantê-las, bem como a recepção mensagens de consistência (contendo *diffs*) que não vão mais ser utilizadas num futuro próximo.

A maioria dos aplicativos escritos para consistência de liberação preguiçosa e de liberação podem ser executados no modelo de escopo sem nenhuma ou com algumas pequenas modificações. Desta forma, os programas desenvolvidos para o TreadMarks, Quarks e JIAJIA podem ser facilmente portados para o Nautilus, bastando-se uma simples troca das primitivas e da inicialização dos programas.

4.3 Protocolo de Coerência Baseado em *Lock*s

O Nautilus adota o protocolo de coerência *lock-based*. A figura 2.8 resume o diagrama de estados para este protocolo.

O JIAJIA[20] somente contém informações das páginas *cached* em cada nó, porque Hu[20] argumenta que isto reduz o *overhead* do sistema. De maneira contrária, no Nautilus, mantêm-se a estrutura de diretório das páginas local, isto é, em todos os nós, pois ela não ocupa muito espaço de memória, já que contém apenas informações relevantes, e ao contrário, auxilia o aumento de *speedup* do sistema.

Os proprietários das páginas no Nautilus não enviam *diffs* para outros nós, conforme o modelo de escopo. Assim, nos nós *home* das páginas, não existe a necessidade de criação e envio dos *diffs*, minimizando-se o número de mensagens para obtenção dos mesmos de vários nós, e também diminuindo a ocupação de memória, o que é mais eficiente do que o mecanismo *lazy diff creation* do TreadMarks.

4.4 Organização da Memória e Mapa de Endereçamento

Cada nó age como uma parte da memória compartilhada. O usuário não pode escolher a distribuição inicial dos dados.

No Nautilus[39, 41], são feitos acessos sem nenhum *overhead* às páginas que pertencem aos nós *homes*. Páginas remotas são trazidas do seu nó *home* e colocadas localmente, portanto ficam no estado *cached*, para subseqűentes futuros acessos. O endereço lógico das páginas é o mesmo em todos os nós, tanto para a página no nó *home*, como para as *cached*, o que elimina translações, permitindo uma visão uniforme da memória compartilhada em todos os nós da rede.

O Nautilus é capaz de suportar memória compartilhada bem maior do que a memória física de cada nó do sistema.

4.5 Sincronização

O Nautilus dispõe dos seguintes elementos para sincronização:

- barreiras;
- semáforos.

A implementação de barreiras no Nautilus é centralizada, isto é, um nó se responsabiliza por receber as mensagens de entrada na barreira. Para minimizar o número de mensagens emitidas na rede, as mensagens de sincronização contém informações de consistência.

As mensagens de entrada de barreira são mensagens emitidas por cada um dos nós participantes do processo, exceto pelo nó que é o servidor de barreira. Estas mensagens são emitidas para o servidor de barreira. Estas mensagens, além de indicar que o nó chegou à barreira, contém uma lista com as páginas que foram modificadas pelo nó (*write-notices* só daquele nó que enviou a mensagem de chegada à barreira). Este mesmo nó, quando todas as mensagens de chegada de barreira já foram recebidas, monta uma mensagem de *broadcast* que contém todos os *write-notices* de todos os nós, inclusive as do servidor. Esta mensagem de *broadcast* para todos os nós, avisando que se trata de uma saída de barreira, é emitida para todos os nós. Estes ao receberem-na, estão sincronizados. Cada nó que recebe a mensagem de *broadcast*, confirma a sua recepção. Se a

mensagem de *broadcast* for perdida, o nó que a emite não recebe as confirmações de tê-la recebido, e portanto as envia novamente após um *timeout*.

Como exemplo, foi dito que as *write-notices* nada mais são do que a lista contendo as páginas modificadas numa seção crítica. As *write-notices* são carregadas junto com as mensagens de *acquire* dos semáforos e na saída das barreiras (modeladas como *acquire*).

De forma análoga, a implementação dos semáforos é centralizada, porém cada conjunto de semáforos tem seu *home*. Para fazer o *acquire* de um semáforo que não é local, o nó que o está pedindo emite uma mensagem para o servidor de semáforos correspondente. No servidor de semáforos, se algum nó já estiver utilizando o *lock* desejado, o pedido é armazenado em uma fila e liberado quando chegar sua vez. Ao se liberar o semáforo, o servidor de semáforos emite uma mensagem para o nó que o está requisitando, mensagem que além de indicar que o nó vai poder utilizar o semáforo e continuar executando a seção crítica, contém as *write-notices*, isto é, as páginas que o nó deve invalidar.

No momento do *release*, o nó emite uma mensagem de *release* daquele semáforo para o servidor de semáforos respectivo. Esta mensagem de *release* contém as *write-notices* do nó que fez o *release*. Quando esta mensagem chega ao servidor, ele irá emitir na próxima liberação, estas últimas *write-notices* recebidas junto com as *write-notices* que ele possui. Este processo é inicializado a cada barreira, ou seja, somente as *write-notices* de um semáforo entre duas barreiras são guardadas.

4.6 Protocolo de Comunicação

O pacote de comunicação adotado para a implementação do Nautilus foi o UDP com *sockets*. O UDP foi escolhido por apresentar menor *overhead* do TCP e porque o Nautilus está sendo desenvolvido e avaliado em uma rede local fechada. Os *sockets* foram escolhidos pois apresentam menor *overhead* em relação aos pacotes MPI e PVM, pois estes são implementados a partir dos primeiros.

4.7 Características do Sistema: tamanho, documentação

O Nautilus contém aproximadamente 3300 linhas de código fonte escritas em linguagem C.

O manual do usuário do Nautilus está apresentado no apêndice B deste trabalho.

Capítulo 5

Resultados Experimentais

O objetivo principal deste capítulo é confrontar o TreadMarks, o JIAJIA e o Nautilus num ambiente composto por hardware de prateleira e sistema operacional aberto. Neste confronto, cujos parâmetros serão detalhados posteriormente, os DSMs serão submetidos a uma série de *benchmarks* utilizados pela maior parte da comunidade acadêmica.

Os comentários relativos aos resultados dos programas apresentados neste capítulo foram colocados de maneira a simplificar a leitura, facilitando sua compreensão. As tabelas e uma análise quantitativa detalhada justificando os comentários estão localizadas no Apêndice A.

5.1 Ambiente de Teste

Os resultados são colhidos numa rede de 16 PCs. Cada PC é equipado com a seguinte configuração:

- Celeron 433 MHz, ;
- 128 MB de memória SDRAM (66 MHz);
- placa de rede fast-ethernet (100 Mbits/s) Intel Ether-Express Pro (conhecida também como Pro/100).

Para medir os *speedups*, a rede foi totalmente isolada de qualquer outra rede externa.

Os PCs foram interconectados através de um switch 3Com 3300 (de 24 portas).

Cada PC está executando o Linux RedHat 6.0 e os experimentos são executados sem processos de quaisquer outros usuários.

5.2 Métrica e Avaliação

Neste trabalho vão ser avaliados os seguintes parâmetros:

- *speedups*;
- número total de mensagens;
- número de mensagens de consistência;
- número total de kbytes;
- número de mensagens contendo página.

Como a avaliação está sendo realizada numa mesma rede de PCs, sob o mesmo sistema operacional, esperam-se obter resultados precisos, homogêneos e justos.

Não faz sentido medir em separado o número de mensagens de sincronização visto que algumas informações de consistência estão 'embutidas' dentro de algumas mensagens de sincronização nos modelos de consistência de liberação preguiçosa e de escopo.

Para a avaliação do Nautilus foram escolhidos dois sistemas DSMs largamente difundidos: o TreadMarks, bastante utilizado pela comunidade científica e conhecido pelos seus excelentes *speedups* e o JIAJIA, de domínio público e também de excelente desempenho. As versões destes softwares são: TreadMarks v1.0.3 e o JIAJIA v2.1. A versão utilizada do Nautilus é a v.0.0.2.

5.3 Escolha da Distribuição de Dados e dos Aplicativos

O JIAJIA permite ao usuário escolher a distribuição dos dados entre os nós da rede, antes de ser executado o programa aplicativo. Porém, como não se dispõe dos fontes do TreadMarks para saber como é feita a distribuição dos dados, optou-se pela distribuição padrão de cada DSM, respectivamente dadas pelas suas primitivas.

Os aplicativos descritos abaixo foram escolhidos porque são alguns dos mais comumente utilizados em avaliações de *speedups* de DSMs.

Os aplicativos EP (NAS), LU (SPLASH-2[86]), MM e SOR (Rice-University) foram avaliados com três parâmetros de entrada diferentes para se observar o efeito da localidade. Para os aplicativos Water (SPLASH-2[86]) e Barnes (SPLASH-2[86]), foi somente utilizado um parâmetro de entrada pois os tempos de execução não eram adequados (suficientemente grandes) para se tomar as pertinentes conclusões a respeito.

Cada programa aplicativo foi executado 5 vezes e foi tirada uma média aritmética para a obtenção dos resultados.

Para demonstrar a compatibilidade do Nautilus, portou-se cada aplicativo avaliado ou do JIAJIA ou do TreadMarks. A escolha foi feita aleatoriamente.

5.3.1 EP (NAS)

O EP (*Embarrassingly Parallel*) do conjunto de *benchmarks* do NAS[20], gera pares de desvios randômicos com um esquema adequado para computação paralela e tabula os números de pares sucessivos. A única comunicação e sincronização deste programa é a soma de uma lista de dez inteiros em uma seção crítica (protegida por um *lock*) no fim do programa. É um *benchmark* que apresenta uma pequena comunicação e um pequeno número de mensagens transmitidas pela rede. No final, cada processador lê a soma e ordena suas chaves[20].

Foi escolhida a versão do DSM JIAJIA e portada para os outros DSMs.

Os parâmetros de entrada aqui utilizados foram: M^{24}, M^{26} e M^{28}.

5.3.2 LU (Blocked - SPLASH-2)

"O kernel LU do SPLASH-2 fatora uma matriz em um produto de duas outras matrizes triangulares: superior e inferior. Para explorar a localidade temporal, a matriz é dividida em nxn submatrizes de bxb blocos. A matriz é fatorada como um array de blocos, permitindo que os blocos sejam alocados contiguamente e inteiramente na memória local dos processadores que a possuem. O kernel LU do SPLASH-2 é um *benchmark* que apresenta a razão computação /comunicação da ordem $O(N^3/N^2)$, aumentando com o tamanho do problema N. Os nós se sincronizam em passos de computação e nenhuma das fases é totalmente paralelizada"[20].

Foi escolhida a versão do DSM JIAJIA e portada aos outros DSMs.

Os parâmetros de entrada aqui utilizados foram matrizes de tamanho: 1024x1024, 1792x1792 e 3072x3072. Estes tamanhos foram escolhidos a partir da observação de outros trabalhos[20] e da memória disponível no sistema, tomando-se o cuidado neste último caso para não ocorrer o fenômeno de *swapping* no TreadMarks. Vale a pena também dizer que os valores escolhidos para os tamanhos das matrizes foram aleatórios.

A distribuição de blocos adotada foi a distribuição contígua que permite que blocos sejam alocados em memória local dos nós que os possuem (*homes*), mesmo que estes blocos não estejam contiguamente alocados na matriz original. O tamanho de bloco selecionado para manter boa distribuição de carga foi de 16 bytes.

5.3.3 MM

"É um simples programa de multiplicação de matrizes com um algoritmo interno. Todas as matrizes estão alocadas em memória compartilhada. Para obter uma boa localidade de dados, o multiplicador é transposto, antes de se efetuar a multiplicação"[20], pois somente são utilizadas duas barreiras.

Este programa apresenta baixa relação comunicação/computação. Somente duas barreiras são utilizadas do começo ao fim do programa. Cada nó irá conter os dados em seu cache, ou seja,

existe um tempo para terminar o *cold startup* e depois nenhuma falta de página mais vai ocorrer até o fim do programa.

Também neste caso, o código fonte escolhido foi o do DSM JIAJIA, sendo portado aos outros DSMs. Os tamanhos das matrizes escolhidos foram: 1024x1024, 1792x1792 e 3072x3072. As considerações sobre os tamanhos são idênticas às do item anterior.

5.3.4 SOR (Rice University)

O aplicativo SOR resolve equações parciais diferenciais (equações de Laplace) com um método de relaxação. Existem duas matrizes, a preta e a vermelha, alocadas em memória compartilhada. Cada elemento da matriz vermelha é calculado como sendo uma média aritmética de elementos da matriz preta e, por sua vez, cada elemento da matriz preta é calculado como sendo uma média aritmética de elementos da matriz vermelha. A comunicação ocorre entre as linhas em uma barreira. Para um determinado número de iterações, cada iteração apresenta duas barreiras[20]. A comunicação não aumenta com o número de processadores e a relação comunicação/computação diminui à medida que o tamanho do problema aumenta.

Neste caso, o código fonte escolhido foi o do DSM TreadMarks e foi portado aos outros DSMs. Os tamanhos das matrizes escolhidos foram: 1024x1024, 1536x1536 e 2048x2048. As considerações sobre os tamanhos são idênticas às dos itens anteriores.

5.3.5 Water (SPLASH-2)

O aplicativo Water *N-Squared* avalia forças e potenciais num sistema de moléculas de água por meio de soluções de equações de Newton para a movimentação de moléculas de água em uma caixa cúbica com condições de contorno periódicas[20].

Neste algoritmo os estados das moléculas são simulados em cada passo, onde forças intra e inter moleculares são computadas. Os arrays de moléculas são divididos em partes iguais associadas aos processadores. A fase que consome mais tempo e que emite mais mensagens acontece na fase de computação das forças intermoleculares, onde cada processador computa e atualiza as forças entre cada uma de suas moléculas e as n/2 moleculas que se seguem. Para reduzir a comunicação uma cópia local das alterações das forças é temporariamente acumulada. Em cada processador existe um *lock* para proteger a atualização das forças pertencentes das respectivas moléculas.

Este *benchmark* somente será avaliado para um único conjunto de parâmetros: 25 passos, 1728 moléculas.

A versão escolhida para o código fonte foi a do TreadMarks e portada aos outros DSMs.

5.3.6 Barnes (SPLASH-2)

O aplicativo Barnes simula a evolução da evolução de corpos sob a influência de forças gravitacionais. O método hierárquico Barnes-Hut é utilizado para reduzir a complexidade do algoritmo.

O aplicativo é dividido em 4 fases em cada passo. As fases são: construção da árvore Barnes-Hut, particionamento de corpos entre os processadores, cálculo das forças e atualização da posição e velocidade dos corpos. A maior parte da computação ocorre na fase de cálculo das forças e as barreiras são o principal mecanismo de sincronização.

Este *benchmark* somente será avaliado para um único conjunto de parâmetros: 16384 corpos. O parâmetro de tolerância escolhido foi de 0.2 para obtenção de melhor *speedup*[20].

A versão escolhida para o código fonte foi a do JIAIA, sendo portada aos outros DSMs avaliados.

5.4 Resultados Comparativos

A avaliação comparativa entre os DSMs será dividida em quatro partes:

1. avaliação comparativa entre os DSMs (parâmetros comentados no item 5.2);
2. avaliação comparativa entre os DSMs, sendo aplicada a técnica de agregação de páginas ao Nautilus e ao JIAJIA;
3. avaliação comparativa entre os DSMs, com o Nautilus usando a técnica de detecção de escrita CO-WD;
4. avaliação comparativa entre os DSMs com ambas as técnicas, agregação de páginas e detecção de escrita CO-WD, aplicadas ao Nautilus

Em cada item, inicialmente existirão algumas uma tabelas comparativas onde serão exibidos parâmetros gerais (mencionados no item 4.8) para **16 nós**. Em seguida, cada *benchmark* será avaliado separadamente, onde serão mostradas as curvas de *speedup*.

5.5 Diferença Não Relevante

Quando a diferença de *speedup* entre os vários DSMs para um determinado *benchmark* **for inferior a 3%**, isto é, quando a diferença não for relevante, **as curvas de *speedup* e a respectiva análise serão feitas no apêndice A**. Conforme será visto posteriormente, constatou-se diferenças não relevantes para os benchmarks **EP** e **Water**.

Somente para salientar, ratificando o que foi dito, **os comentários relativos aos resultados dos programas apresentados neste capítulo foram colocados de maneira a simplificar a leitura, facilitando sua compreensão. As tabelas e uma análise quantitativa detalhada justificando os comentários estão localizadas no Apêndice A.**

DSM/parâmetro	*page fault* (ms)	barreira (ms)	semáforo (ms)
Tmk	4	2	3
JIA	4	2	3
Naut	4	2	3

Tabela 5.1: Alguns parâmetros básicos de comparação

5.6 Avaliação comparativa entre os DSMs

As tabelas que se seguem vão conter os valores medidos dos parâmetros gerais mais importantes quando se comparam vários DSMs.

Para a compreensão das tabelas e gráficos que se seguem, é necessário definir os seguintes parâmetros:

- Tmk: referente ao TreadMarks;
- JIA: referente ao JIAJIA;
- Naut: referente ao Nautilus;
- t(1): tempo em segundos para a execução do correspondente programa seqüencial usando primitivas padrão de alocação de memória (**malloc()** do C);
- t(16): tempo em segundos para execução em 16 nós usando algum DSM;
- Sp: *speedup* com 16 nós, calculado pela divisão do tempo seqüencial t(1) pelo tempo de execução em 16 nós (t(16)); Sp(16) = $\frac{t(1)}{t(16)}$
- msgs: número total de mensagens;
- kB: número total de kbytes transferidos;
- page: número total de mensagens contendo página (ou simplesmente de páginas, já que cada mensagem contendo página contém uma página);
- SIGS: número de sinais SIGSEGV (violação de segmentação);
- *diffs*: número de mensagens de consistência (que podem conter vários *diffs*);
- cel: o ambiente é uma rede de Celerons;
- 4096: o DSM está utilizando página de 4kB;
- 8192: o DSM está utilizando página de 8kB (agregação de duas páginas);

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
t(1)	46.7	187.3	750.0	210.2	1148	5847	75.1	404.2	2063	2.9	6.3	11.4	8027	320
t(16).Tmk.4096	2.99	11.93	47.76	14.04	74.34	371.9	6.98	33.9	147.3	0.93	1.49	3.52	519.4	29
t(16).JIA.4096	3.13	12.04	47.74	14.83	76.82	380.0	10.3	37.2	169.2	2.18	4.61	10.45	546.0	72
t(16).Naut.4096.n	3.58	14.14	56.47	14.98	76.66	381.3	5.67	30.2	140.4	0.89	1.67	2.57	522.0	27
Sp(16).Tmk.4096	15.67	15.70	15.70	14.97	15.44	15.72	10.8	11.9	14.0	3.07	4.25	3.22	15.45	11.43
Sp(16).JIA.4096	14.95	15.56	15.71	14.18	14.95	15.39	7.29	10.9	12.2	1.31	1.37	1.09	14.70	4.44
Sp(16).Naut.4096.n	13.07	13.24	13.28	14.03	14.97	15.33	13.3	13.4	14.7	3.21	3.79	4.41	15.38	11.85

Tabela 5.2: Tempos e *speedup*s para 16 nós

- 16384: o DSM está utilizando página de 16kB (agregação de quatro páginas);
- 32768: o DSM está utilizando página de 32kB (agregação de oito páginas);
- .n: DSM não está utilizando a técnica de detecção de escrita tradicional;
- .c: DSM está utilizando a técnica de detecção de escrita CO-WD (*cache only write detection technique*);
- -: quando este sinal for encontrado, significa que a execução do aplicativo não terminou normalmente.

Alguns parâmetros básicos para fins de comparação são mostrados na tabela 5.1. O parâmetro *page fault* é o tempo médio de uma *page fault* para os diversos programas que foram executados neste experimento. Da mesma forma, o parâmetro barreira é o tempo médio para efetuar uma primitiva de barreira com intuito somente de sincronizar os vários processadores, sem levar em conta as operações de consistência envolvidas. Analogamente, o parâmetro semáforo é o tempo médio para efetuar uma primitiva de semáforo sem levar em conta as operações envolvidas.

Como se pode perceber pela tabela 5.1, os DSM apresentam resultados semelhantes. As diferenças nos *benchmarks* individuais são devidas a uma melhor adequação das características do DSM ao caso particular de cada código, embora tenham tenham princípios e codificação bastante diferentes.

5.6.1 LU (SPLASH-2)

Os *speedups* do aplicativo LU podem ser vistos na figura 5.1, onde são mostrados os *speedups* dos três DSMs para três diferentes parâmetros de entrada 1024, 1792 e 3072 (figura 5.1). Os *speedups* aumentam com o aumento do tamanho do problema (parâmetro de entrada) pois a razão entre computação/comunicação do LU é do tipo $O(N^3/N^2)$[20].

Como se pode perceber pela figura 5.1, o comportamento geral dos gráficos é o seguinte: até 8 nós o JIAJIA é mais rápido que os outros, enquanto que para mais de 8 nós, o TreadMarks

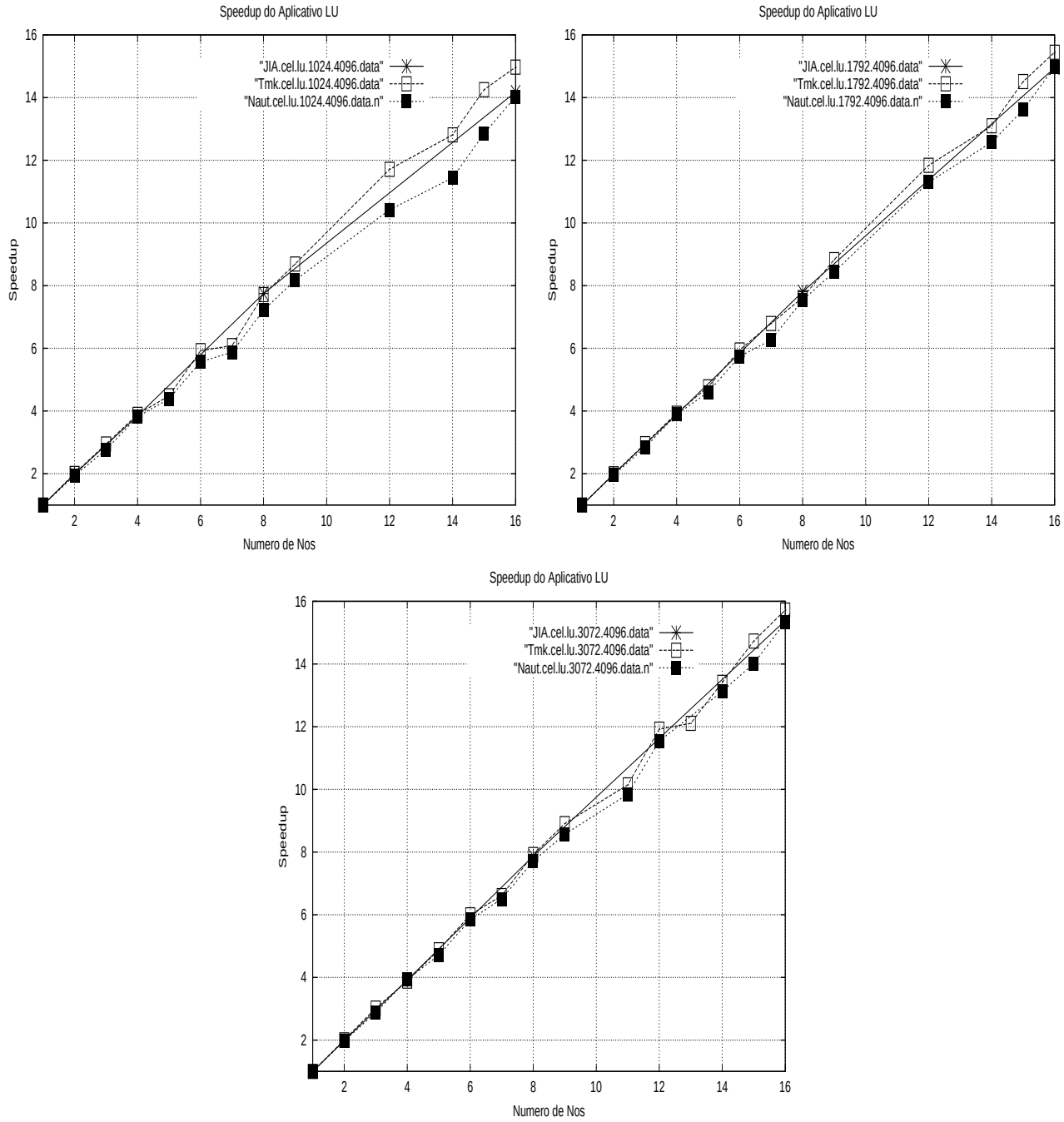

Figura 5.1: *Speedups* do aplicativo LU (SPLASH-2) - tamanhos 1024, 1792 e 3072

é mais rápido que o JIAJIA e este, mais rápido que o Nautilus, exceto para tamanho de matriz igual a 1792. Para 1792, a distribuição de dados desfavorece o Nautilus, o que acarreta uma perda de desempenho. Com 16 nós, o desempenho do Nautilus fica praticamente o mesmo do JIAJIA. Como se pode perceber na tabela 5.2, em valores numéricos, para 16 nós, a diferença de *speedup* entre o TreadMarks e o JIAJIA e entre TreadMarks e o Nautilus chega a respectivamente 5,3% e 6,3% para tamanho de matriz 1024, 3,2% e 3,0% para 1792 e, 2,1% e 2,5% para 3072. Deve-se observar também que à medida que se aumenta o tamanho das matrizes de entrada, mais altos ficam os *speedups* devido à melhora da localidade dos programas.

Outro ponto interessante que deve ser comentado é que o JIAJIA somente pode ser executado com número de nós potência de 2; portanto se alguns pontos dos gráficos de *speedup* do TreadMarks e Nautilus fossem desprezados não se perceberia o comportamento oscilatório das curvas, comportamento esse devido a divisão de blocos de matrizes em número de nós não múltiplos de potência de 2.

As razões que justificam o melhor *speedup* do TreadMarks em relação ao JIAJIA são: i)apesar do JIAJIA transmitir no global menos mensagens que o TreadMarks e menos kbytes, o JIAJIA transmite um maior número de mensagens relativas a páginas que o TreadMarks, que por sua vez transmite um número maior de *diffs*, graças à adoção dos modelos de consistência de escopo (*home-based*) e o modelo de consistência de liberação preguiçosa respectivamente; ii) sobrecarga dos nós por pedidos de página no JIAJIA, o que acaba aumentando o tempo de espera.

Pode-se dizer que o TreadMarks apresenta um maior desempenho em relação ao Nautilus devido predominantemente a maior utilização de mensagens relativas a páginas pelo Nautilus versus a utilização de *diffs* do TreadMarks, reflexo direto dos modelos de consistência adotados por cada um deles. A maior utilização de páginas pelo Nautilus acaba resultando numa maior quantidade de kbytes transmitida, acabando por ocupar uma banda maior. Uma observação complementar é que embora o JIAJIA e o Nautilus apresentem o mesmo modelo de consistência (escopo), diferem na distribuição de dados, conseqűentemente no falso compartilhamento, o que neste caso acabou prejudicando o desempenho do Nautilus. Não se deve esquecer também as diferentes codificações.

5.6.2 MM

Os *speedups* do aplicativo MM podem ser vistos na figura 5.2, onde são mostrados os *speedups* dos três DSMs para três diferentes parâmetros de entrada 1024, 1792 e 3072 (respectivamente figuras 5.2 a, b e c).

Como comportamento geral que pode ser percebido pela figura 5.2, o Nautilus é mais rápido que o TreadMarks e que o JIAJIA para os vários parâmetros de entrada (1024, 1792 e 3072). Como se pode perceber pela tabela 5.2, em valores numéricos, para 16 nós, a diferença de *speedup* entre o Nautilus e TreadMarks chega respectivamente a 18,8%, e entre Nautilus e JIAJIA a 45,0% para tamanho de matriz 1024, a 10,9%, e a 18,7% para 1792, a 4,8% e a 17,0% para 3072. Uma outra observação é que à medida que se aumenta o tamanho das matrizes de entrada, melhores

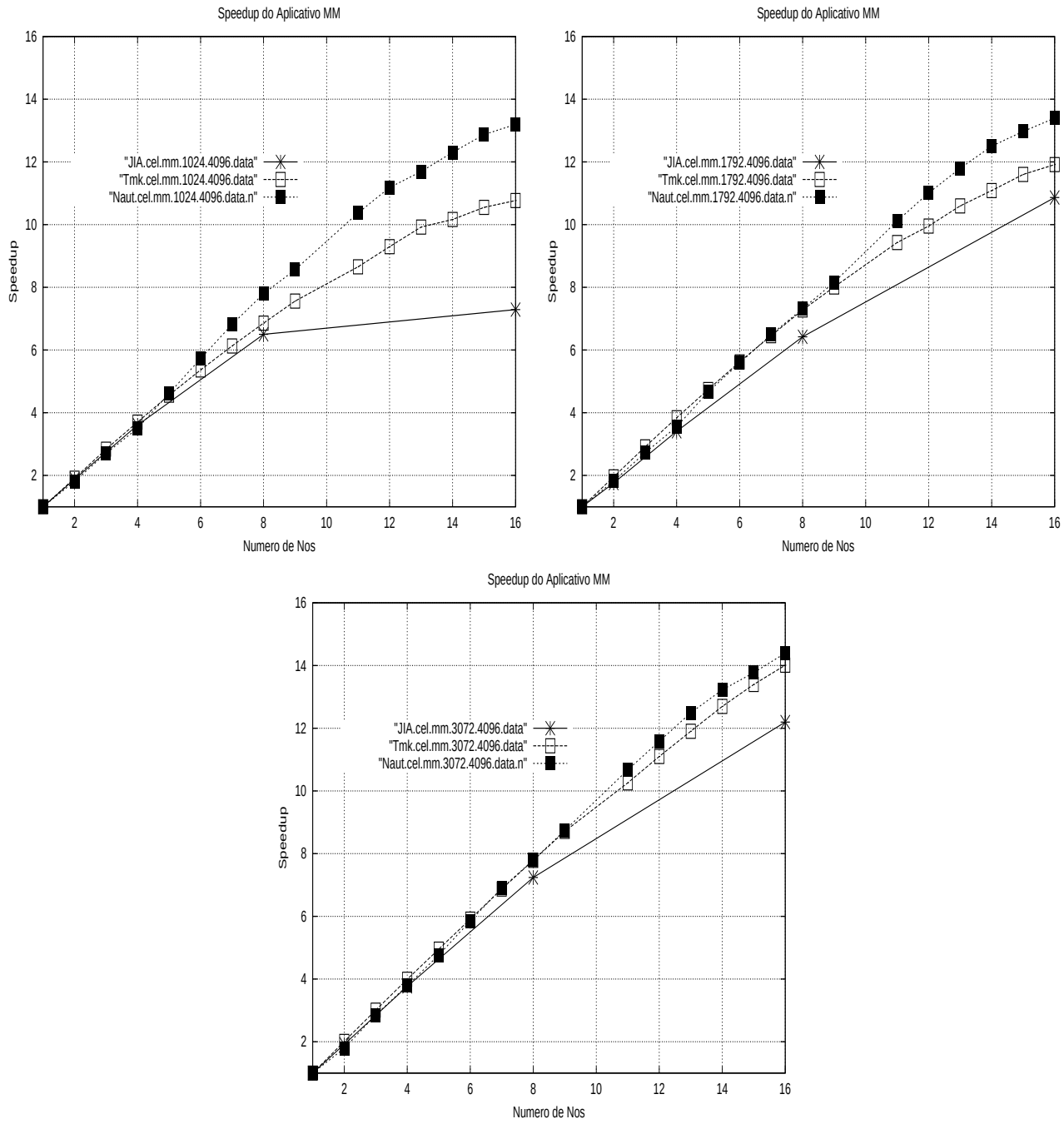

Figura 5.2: *Speedups* do aplicativo MM - tamanhos 1024, 1792 e 3072

ficam os *speedups* devido à melhora da localidade da multiplicação de matrizes (como comentado anteriormente, é um programa que tem uma relação comunicação/computação bem baixa, já que existem somente duas barreiras), e também mais próximas ficam as curvas de *speedup*.

Pode-se dizer que o Nautilus apresenta um melhor *speedup* em relação ao TreadMarks devido ao menor número global de mensagens transmitido, menor número de mensagens de páginas e também menor quantidade de kbytes. A distribuição de dados adotada pelo Nautilus, permitiu um tempo de *cold-startup* menor, desta forma resultando num número menor de mensagens relativas a páginas, conseqüentemente um menor número global de páginas, o que minimizou a utilização da rede, permitindo um melhor *speedup*. O *multithreading* e a não utilização do SIGIO também ajudaram a diminuir os *overheads* do Nautilus.

Confrontando o TreadMarks com o JIAJIA, apesar do JIAJIA transmitir menor número de mensagens, de kbytes e de páginas, o JIAJIA apresentou um *speedup* pior, devido a um possível problema de implementação.

5.6.3 SOR (Rice University)

Na figura 5.3, podem ser vistos os *speedups* do aplicativo SOR para os vários DSMs envolvendo diversos parâmetros de entrada (1024, 1536 e 2048). Pode-se perceber um aumento do *speedup* com o aumento da matriz de entrada pois, segundo Hu[20], a cada iteração existem $O(N^2)$ operações em ponto flutuante e O(N/tamanho da matriz) de faltas de página.

Os *speedups* do JIAJIA não foram colocados na figura 5.3 porque não foram muito usuais, não sendo considerados para efeitos de comparação.

Pode-se dizer que o Nautilus é mais rápido que o JIAJIA para os tamanhos de matriz de 1024 e 2048, mas não para o caso de 1536. Pode-se justificar o melhor *speedup* graças ao menor número de mensagens transmitidas, menor número de kbytes, menor número de mensagens de páginas e menor número de mensagens de *diffs*. Estes parâmetros foram menores devido ao modelo de consistência adotado pelo Nautilus (de escopo), bem como a distribuição de dados que acaba minimizando o efeito do falso compartilhamento e minimizando o tempo de *cold-startup*. Para o caso de tamanho 1536, a distribuição de dados causa um maior efeito do falso compartilhamento o que prejudica o desempenho do Nautilus.

O *multithreading* e a não utilização do SIGIO também ajudaram a minimizar os *overheads* do Nautilus.

5.6.4 Barnes (SPLASH-2)

Na figura 5.4 podem ser vistos os *speedups* dos três DSMs sob o aplicativo Barnes para um único conjunto de parâmetros de entrada: 16384 corpos gravitacionais.

O Nautilus apresentou *speedups* melhores que o TreadMarks devido a um número bem maior de mensagens transmitido pelo TreadMarks, a maior parte mensagens relativas a *diffs*, originadas

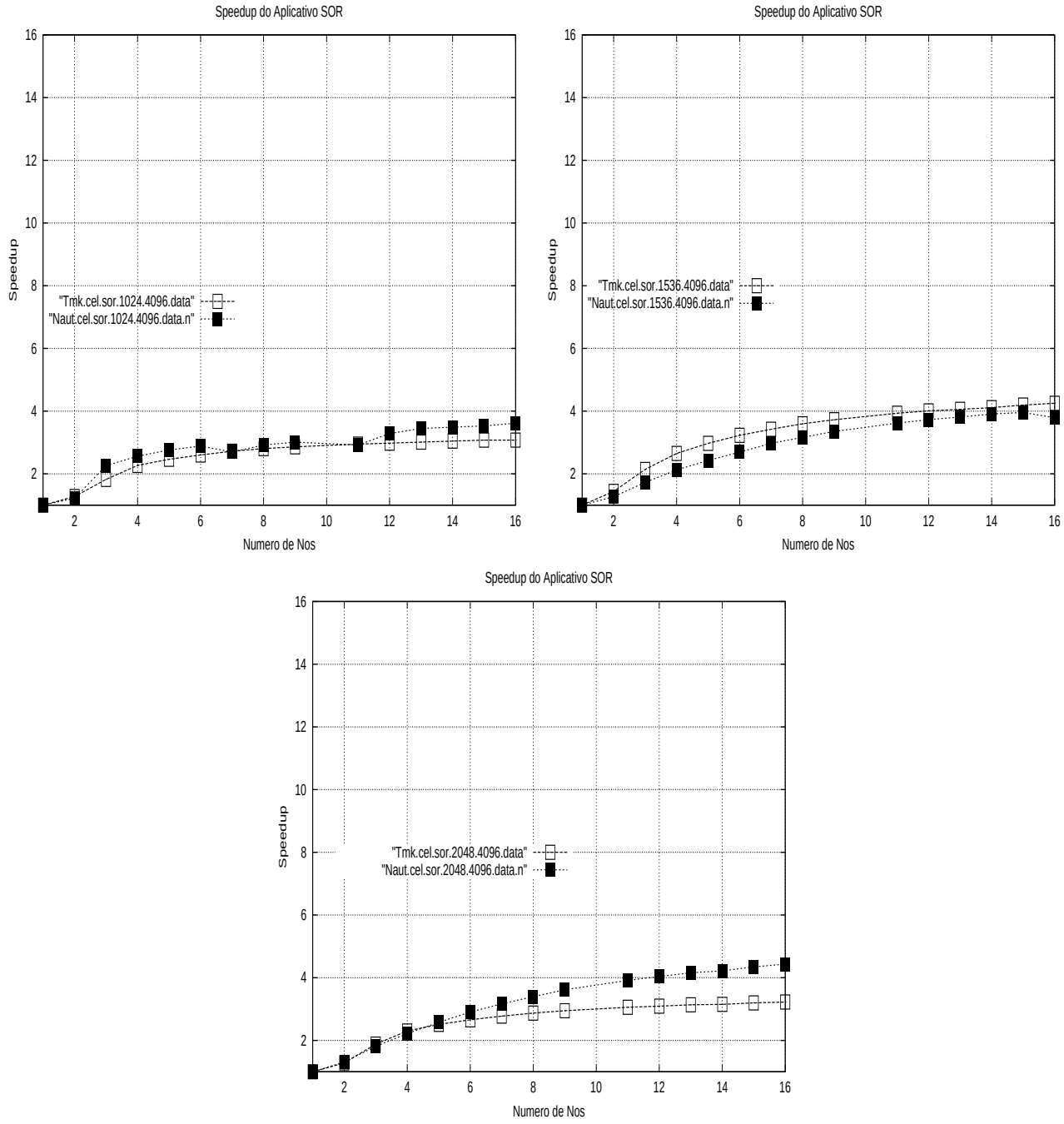

Figura 5.3: *Speedups* do aplicativo SOR (Rice-University) - tamanhos 1024, 1536 e 2048

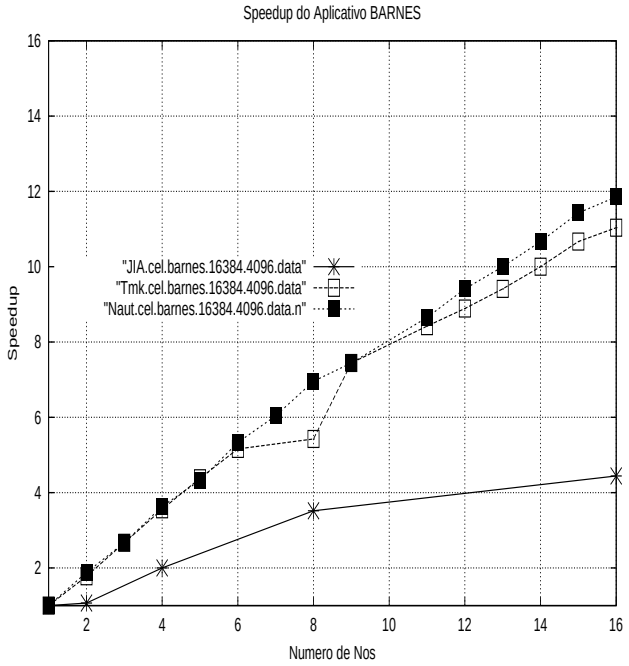

Figura 5.4: *Speedups* do aplicativo Barnes (SPLASH-2) - 16384 corpos

pela adoção do modelo de consistência de liberação preguiçosa. O efeito direto de haver um grande número de *diffs* é a maior quantidade de memória utilizada, maior complexidade para a manutenção de tabelas de *diffs*, bem como maiores tempos de geração e aplicação dos mesmos.

A diferente implementação, que inclui o *multithreading* e a eliminação do SIGIO, bem como uma distribuição de dados que minimiza o tempo de *cold-startup* e o número de páginas transmitidas, favoreceram o Nautilus quando comparado ao JIAJIA.

5.6.5 Conclusões Gerais

Nesta comparação, os DSMs TreadMarks, JIAJIA e Nautilus foram comparados em termos de tempo de execução, *speedups*, número de mensagens transmitidas na rede, número de kbytes transferidos, número de mensagens contendo páginas, número de SIGSEGVs e número de mensagens de *diffs*, para diversos programas aplicativos.

Para os programas EP e Water, os *speedups* dos DSMs empatam devido às características inerentes destes programas, respectivamente devido a pouca transmissão de dados do aplicativo EP e a alta sincronização do Water.

Para o aplicativo LU, o TreadMarks apresentou melhor desempenho em relação ao JIAJIA e Nautilus principalmente graças a seu modelo de consistência.

Para os programas MM, SOR e Barnes, o menor tempo de *cold-startup* e melhor distribuição de dados, auxiliados pelo modelo de consistência, permitiram ao Nautilus apresentar melhores *speedups* que os demais.

programas->	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
t(1)	46.73	187.28	750.03	210.22	1148.20	5847.02	75.13	404.20	2063.31	2.86	6.33	11.35	8027.54	320
t(16).Tmk.4096	2.99	11.93	47.76	14.04	74.34	371.88	6.98	33.93	147.28	0.93	1.49	3.52	519.43	29
t(16).JIA.4096	3.13	12.04	47.74	14.83	76.82	380.04	10.31	37.21	169.17	2.18	4.61	10.45	546.01	72
t(16).JIA.8192	3.05	11.99	47.88	14.63	76.43	380.70	7.84	32.55	153.67	1.94	2.90	5.79	543.39	67
t(16).JIA.16384	3.09	12.03	47.89	14.66	76.13	380.55	7.01	31.36	149.61	2.41	3.63	5.16	549.85	65
t(16).JIA.32768	3.16	12.01	47.95	14.84	76.21	379.86	6.50	30.99	148.51	3.48	5.64	9.38	568.27	63
t(16).Naut.4096.n	3.58	14.14	56.47	14.98	76.66	381.3	5.67	30.25	140.36	0.89	1.67	2.57	522.01	27
t(16).Naut.8192.n	3.59	14.15	56.47	15.10	76.38	381.1	5.69	30.26	140.37	0.62	1.24	2.11	577.71	26
t(16).Naut.16384.n	3.61	14.16	56.48	15.63	77.07	381.9	5.62	30.25	140.36	0.53	0.94	1.60	539.98	26
t(16).Naut.32768.n	3.65	14.20	56.48	17.50	79.22	383.1	5.56	30.24	140.37	2.51?	2.57	1.38	539.14	28
Sp(16).Tmk.4096	15.67	15.70	15.70	14.97	15.44	15.72	10.76	11.91	14.00	3.07	4.25	3.22	15.45	11.43
Sp(16).JIA.4096	14.95	15.56	15.71	14.18	14.95	15.39	7.29	10.86	12.20	1.31	1.37	1.09	14.70	4.44
Sp(16).JIA.8192	15.33	15.61	15.67	14.37	15.02	15.36	9.58	12.42	13.43	1.47	2.18	1.96	14.77	4.78
Sp(16).JIA.16384	15.15	15.57	15.66	14.34	15.08	15.36	10.72	12.89	13.79	1.18	1.74	2.20	14.60	4.92
Sp(16).JIA.32768	14.79	15.47	15.64	14.17	15.06	15.39	11.56	13.04	13.89	0.82	1.21	1.21	14.13	5.08
Sp(16).Naut.4096.n	13.07	13.24	13.28	14.03	14.97	15.33	13.25	13.36	14.70	3.21	3.79	4.41	15.38	11.85
Sp(16).Naut.8192.n	13.04	13.24	13.28	13.92	15.03	15.34	13.20	13.36	14.70	4.61	5.11	5.37	13.89	12.31
Sp(16).Naut.16384.n	12.97	13.23	13.28	13.45	14.90	15.31	13.37	13.36	14.70	5.40	6.73	7.09	14.87	12.31
Sp(16).Naut.32768.n	12.82	13.19	13.28	12.01	14.49	15.26	13.51	13.37	14.70	1.13?	2.46	8.22	14.89	11.43

Tabela 5.3: Tempos e *speedups* para 16 nós - Técnica de Agregação de Páginas

5.7 Técnica de Agregação de Páginas

Para a aplicação da técnica de agregação de páginas, o tamanho de página foi alterado e novas versões de biblioteca foram geradas para o Nautilus e para o JIAJIA, cujos fontes são disponíveis. Assim, foram geradas quatro versões para o JIAJIA e quatro para o Nautilus, com páginas de tamanho 4kB, 8kB, 16kB e 32kB. Não foram avaliados tamanhos superiores a 32kB porque isto acarretaria um grande *overhead* no empacotamento e desempacotamento de mensagens visto que o buffer do UDP é de 64 kB.

Infelizmente, não estavam disponíveis os fontes do TreadMarks para que pudessem ser geradas quatro novas versões de sua biblioteca com os tamanhos de páginas alterados.

Para efeito de facilitar a compreensão do leitor na leitura dos gráficos, os efeitos da técnica de agregação foram separados em dois casos:

1. efeito da técnica sobre o DSM JIAJIA;

2. efeito da técnica sobre o DSM Nautilus.

Os *speedups* do TreadMarks foram repetidos em ambos os gráficos para efeito de referência.

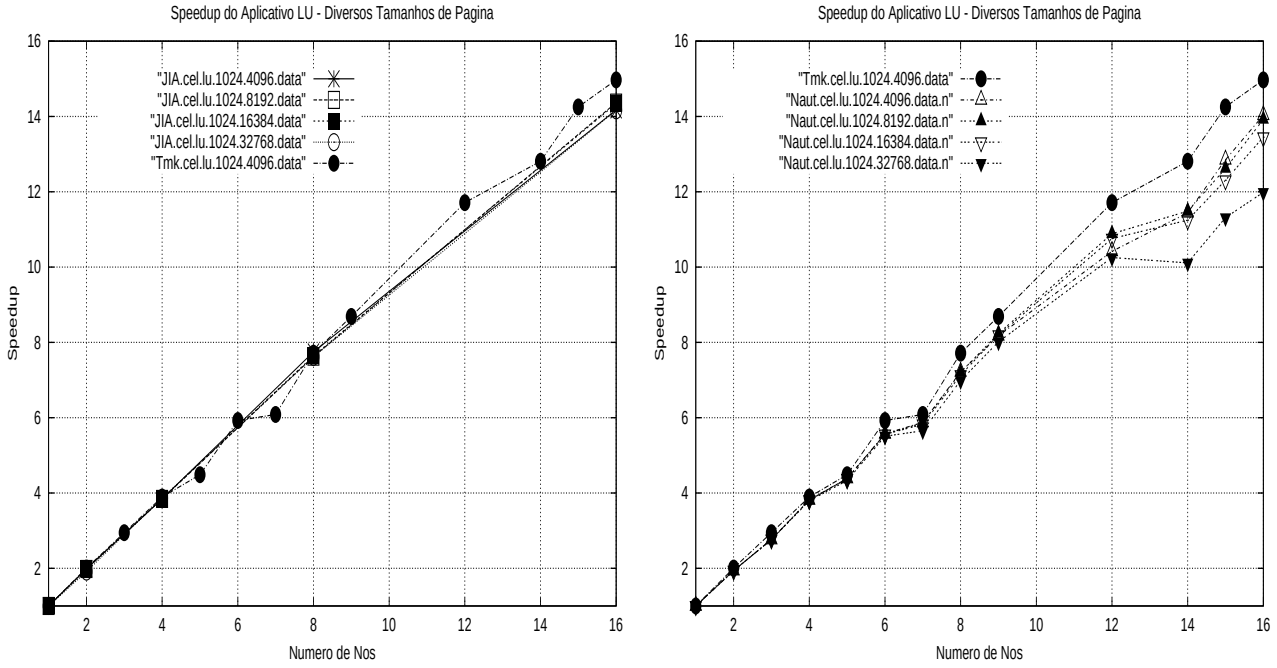

Figura 5.5: *Speedups* do aplicativo LU (SPLASH-2) - tamanho 1024 - Técnica de Agregação de Páginas

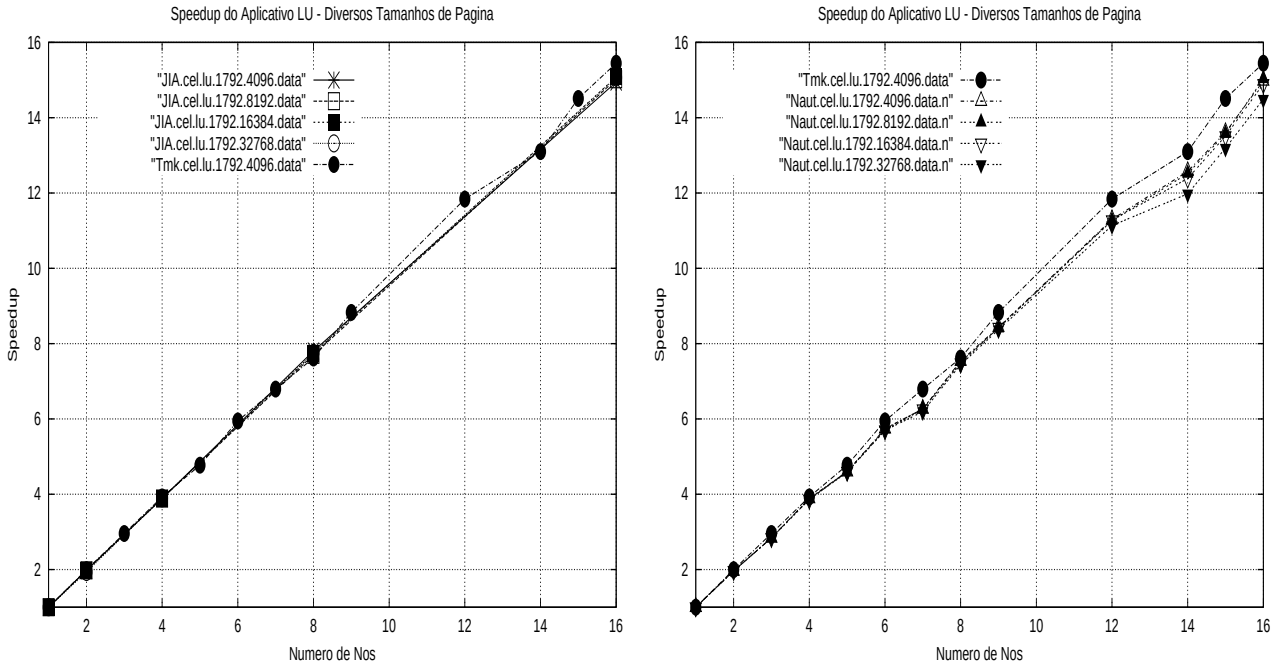

Figura 5.6: *Speedups* do aplicativo LU (SPLASH-2) - tamanho 1792 - Técnica de Agregação de Páginas

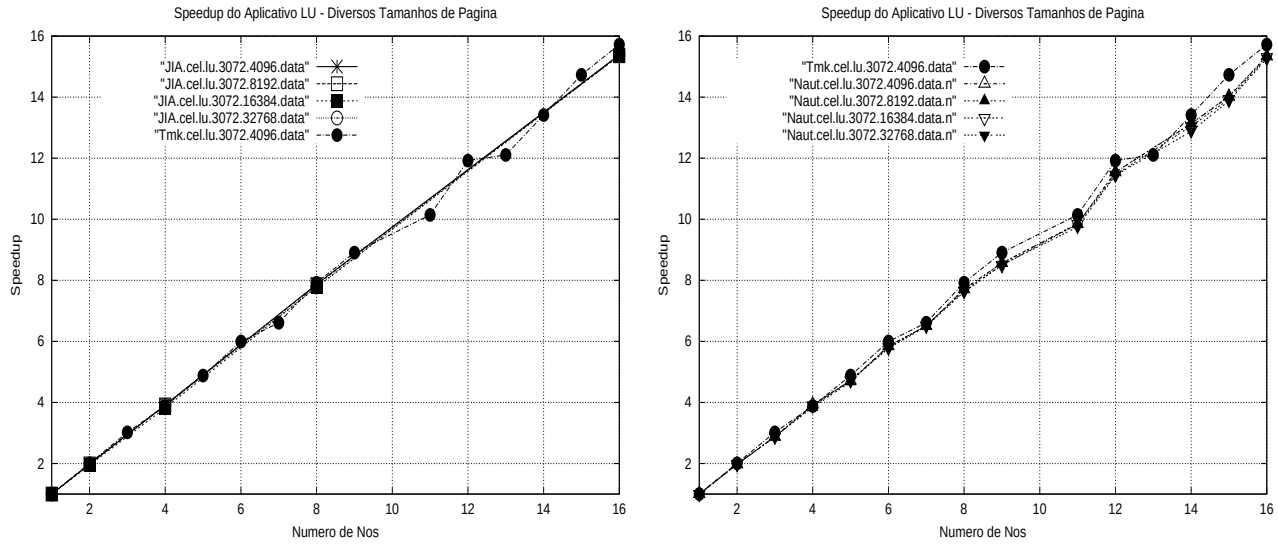

Figura 5.7: *Speedups* do aplicativo LU (SPLASH-2) - tamanho 3072 - Técnica de Agregação de Páginas

5.7.1 LU (SPLASH-2)

Nas figuras 5.5, 5.6 e 5.7 estão plotados os gráficos de *speedups* para o LU aplicando-se a técnica de agregação de páginas ao JIAJIA e Nautilus. Como se pode perceber, neste *benchmark* a técnica de agregação de página alterou significativamente, os *speedups* em alguns casos.

Avaliando a técnica de agregação no DSM JIAJIA, houve um ligeiro aumento de até 1,2% de *speedup* para os tamanhos de 8kB e 16kB e permanecendo no mesmo nível de *speedup* quando foi utilizada página de 32kB, como pode ser visto nas figuras 5.5, 5.6 e 5.7, e na tabela 5.3. O número de mensagens foi reduzido, porém em contrapartida o número de kbytes aumentou. Mesmo com um menor número de mensagens, o tamanho maior das mensagens aumentou a banda utilizada na rede, bem como o tempo de espera por uma página, o não permitiu uma melhora substancial de *speedup*. E também, conforme esperado, o número de mensagens de páginas e o número de SIGSEGVs foram reduzidos de 4 vezes, inversamente proporcional ao aumento do tamanho da página (8 vezes), exceto para o tamanho de 1792, quando o número de SIGSEGVs foi aumentado devido ao aumento do efeito do falso compartilhamento.

Para o Nautilus, à medida que aumentou-se o tamanho de página, ocorreu uma redução de *speedup* que chegou até 14,4% (tabela 5.3 e pelas figuras 5.5, 5.6 e 5.7). O número de mensagens sofreu consideráveis reduções, conseqüência da redução do número de mensagens de páginas pelo aumento de seu tamanho. Outra conseqüência esperada foi a redução do número de SIGSEGVs, pois menos falhas de páginas ocorreram porque menos acessos são feitos às páginas, que apresentam tamanho maior. Porém, como as mensagens de páginas aumentaram de tamanho, o falso compartilhamento também aumentou, sendo responsável pelo aumento da quantidade de kbytes. Outra conseqüência foi o aumento do tempo para responder ao pedido de uma página. Estes dois últimos fatores aliados foram responsáveis pela redução de desempenho do Nautilus à medida que se aumentou o tamanho da página.

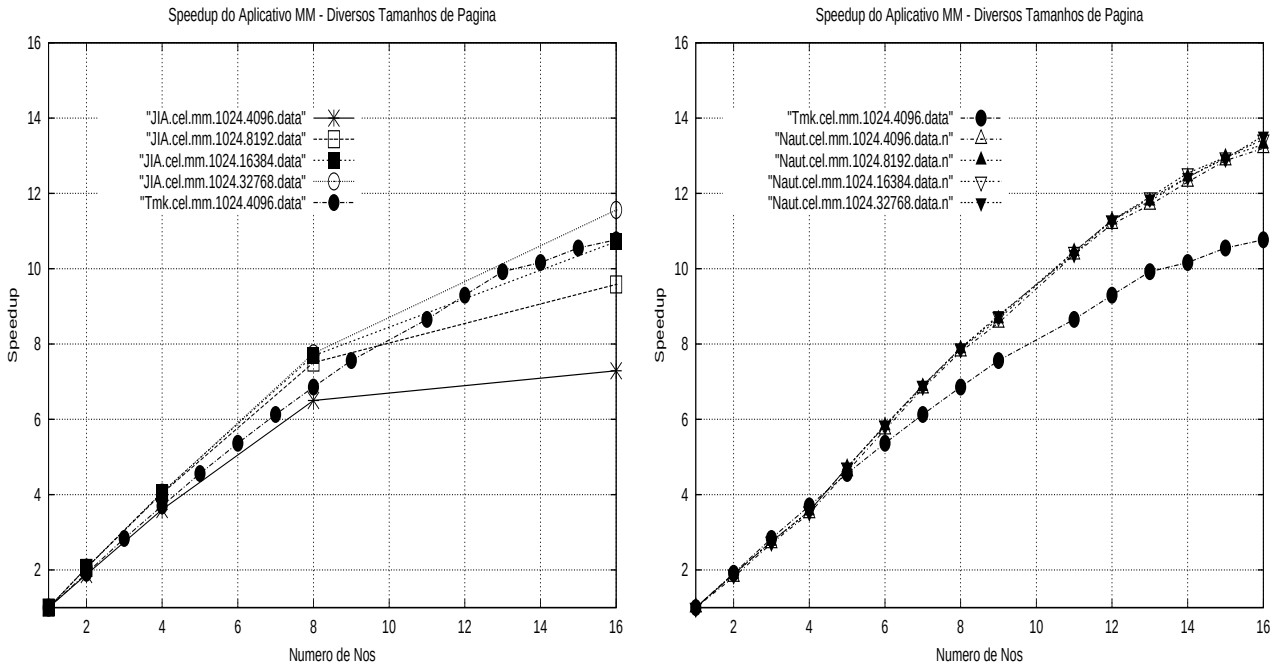

Figura 5.8: *Speedups* do aplicativo MM - tamanho 1024 - Técnica de Agregação de Páginas

5.7.2 MM

Nas figuras 5.8, 5.9 e 5.10 estão apresentadas as curvas de *speedup* do aplicativo MM para os DSMs quando se aplicou a técnica de agregação de páginas ao JIAJIA e ao Nautilus.

Pode-se notar através da tabela 5.3 e pelas figuras mencionadas no parágrafo acima que os *speedups* do JIAJIA aumentam consideravelmente à medida que os tamanhos de página são incrementados. Outras conseqüências da aplicação da técnica de agregação foram uma considerável redução do número de mensagens, devido à redução da parcela de número de mensagens relativas a páginas (pelo aumento do tamanho), acompanhada de uma redução do número de kbytes. A redução do número de mensagens relativas à paginas foi inversamente proporcional ao aumento do tamanho, ambos em torno de 8 vezes. O número de SIGSEGVs também acompanhou a redução do número de mensagens de páginas, conforme esperado. Concluindo, graças à redução do número de mensagens (redução da parcela relativa a páginas) e do número de kbytes, a técnica de agregação permitiu o aumento de desempenho do JIAJIA.

Como se pode verificar por meio das figuras 5.8, 5.9 e 5.10 e da tabela 5.3, os *speedups* do Nautilus praticamente não mudam com o incremento do tamanho de página. Já que quase 100% das mensagens do Nautilus são relativas a páginas neste aplicativo, a técnica de agregação atuando sobre o número de mensagens de páginas, fez com que o número de mensagens global fosse bastante reduzido. Outro efeito esperado foi a redução do número de SIGSEGVs. Apesar das grandes reduções do número de mensagens, na maior parte constitiídas de páginas, o tempo para atender a pedidos de páginas aumentou, ocorrendo uma sobrecarga por pedidos de uma mesma página. Aumentou também o tempo de tratamento (empacotamento e desempacotamento) de uma mensagem de página.

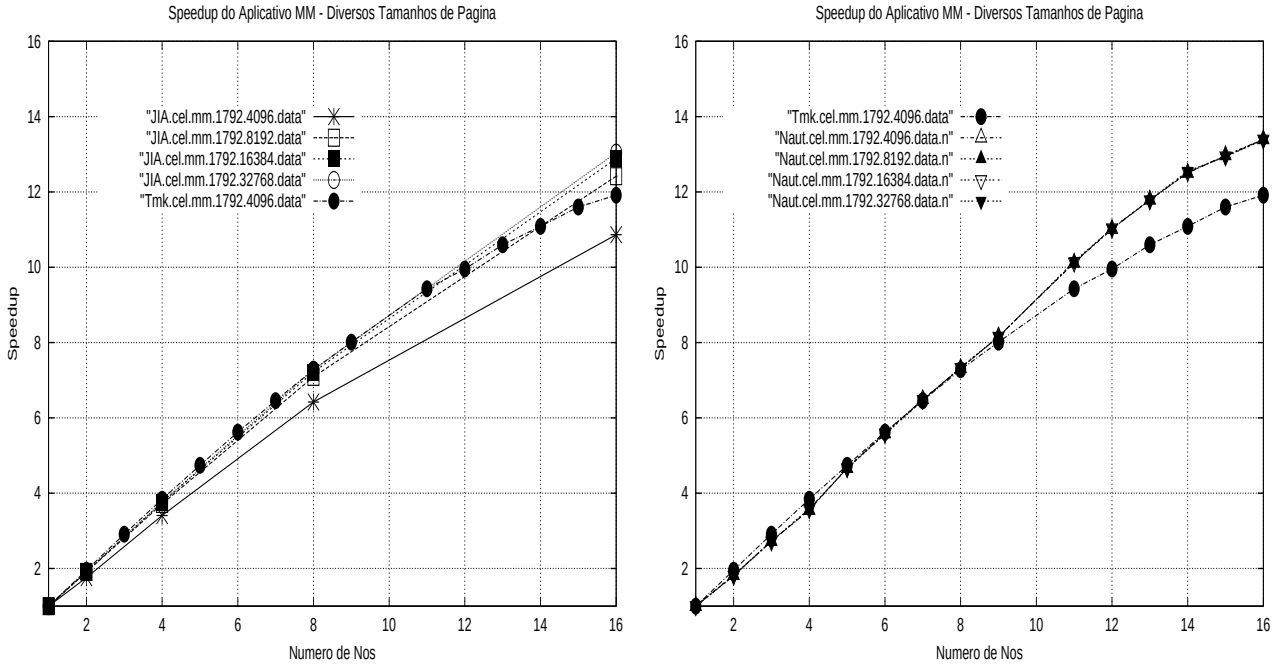

Figura 5.9: *Speedups* do aplicativo MM - tamanho 1792 - Técnica de Agregação de Páginas

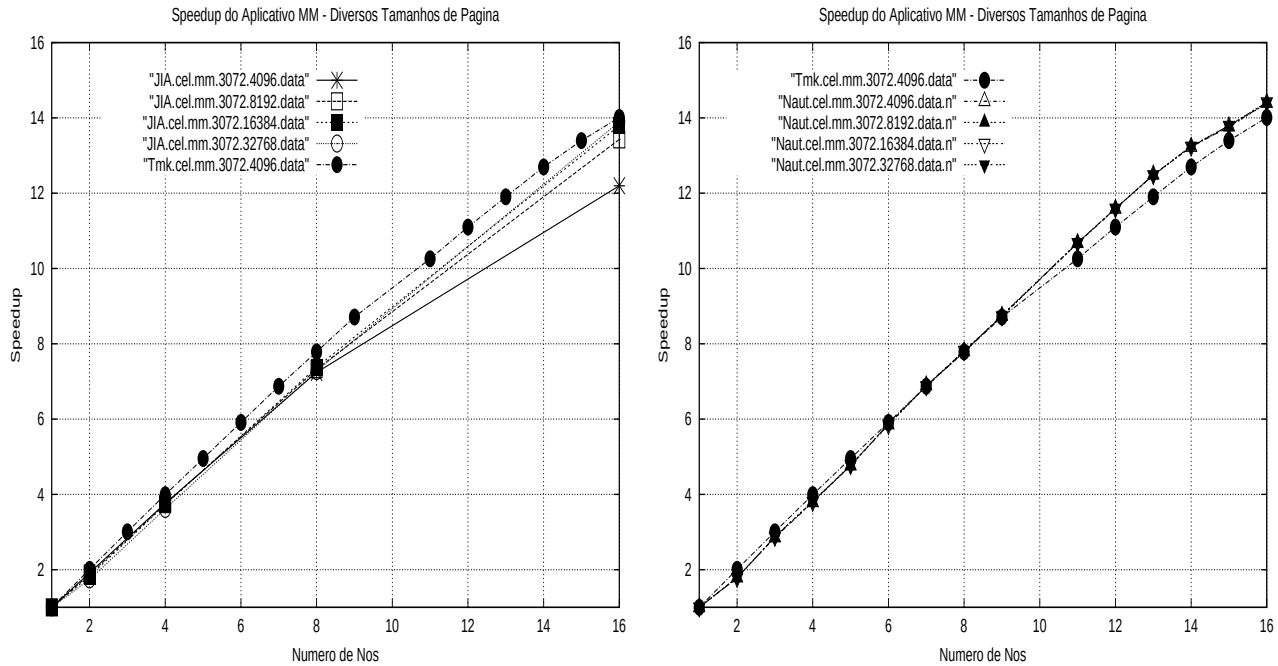

Figura 5.10: *Speedups* do aplicativo MM - tamanho 3072 - Técnica de Agregação de Páginas

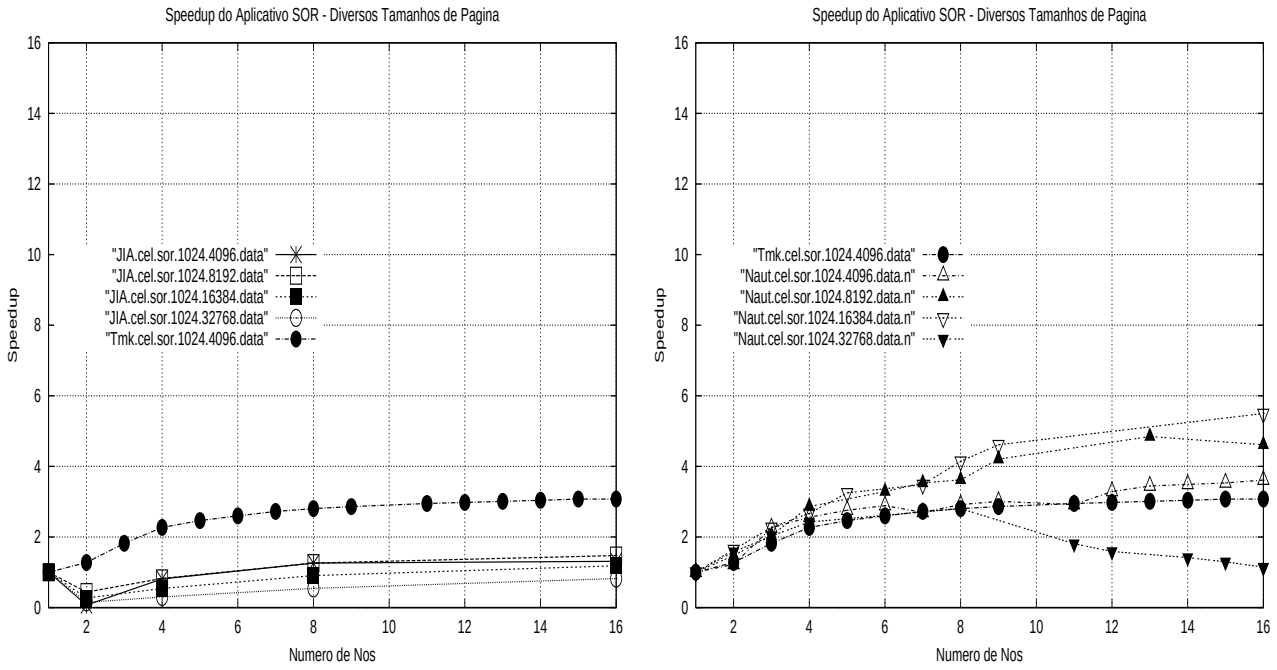

Figura 5.11: *Speedups* do aplicativo SOR (Rice University)- tamanho 1024 - Técnica de Agregação de Páginas

5.7.3 SOR (Rice University)

Nas figuras 5.11, 5.12 e 5.13 podem ser vistos os *speedups* para o aplicativo SOR com a aplicação da técnica de agregação de páginas ao Nautilus, pois conforme foi dito anteriormente, devido aos *speedups* não muito usuais (algum provável problema de implementação), os *speedups* do JIAJIA bem como os outros parâmetros que estão sendo analisados não serão comparados nem confrontados com os demais.

Nota-se que nas figuras mencionadas no parágrafo acima que para as matrizes de entrada de 1024 e 1536, e para páginas de 16kB, os *speedups* do Nautilus aumentam, ao contrário do que ocorre para páginas de 32kB. Como resultado da aplicação da agregação, o número de mensagens foi bastante reduzido devido à redução do número de mensagens de páginas. Acompanhando esta redução, o número de SIGs foi reduzido. Contrariamente aos anteriores, ocorreu o aumento da quantidade de kbytes, por causa do maior efeito do falso compartilhamento. O aumento do tamanho de página causa o aumento do tempo de espera por uma página e também pode causar filas de espera por páginas, o que acaba sobrecarregando o sistema e diminuindo o *speedup* do Nautilus para páginas de 32kB.

5.7.4 Barnes (SPLASH-2)

Na figura 5.14 podem-se ver os *speedups* do aplicativo Barnes, quando submetido aos vários DSMs avaliados, com o JIAJIA e o Nautilus empregando a técnica de agregação de páginas.

Observando-se a figuras 5.14 e a tabela 5.3, percebe-se um aumento do *speedup* do JIAJIA

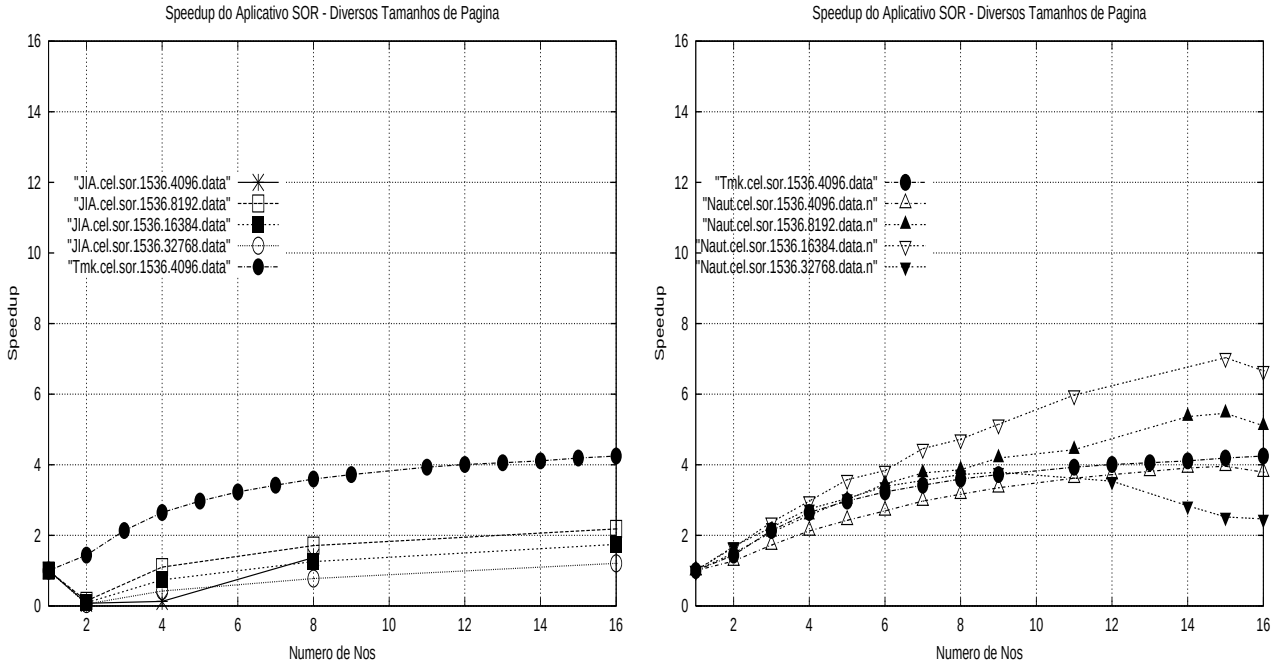

Figura 5.12: *Speedups* do aplicativo SOR (Rice University)- tamanho 1536 - Técnica de Agregação de Páginas

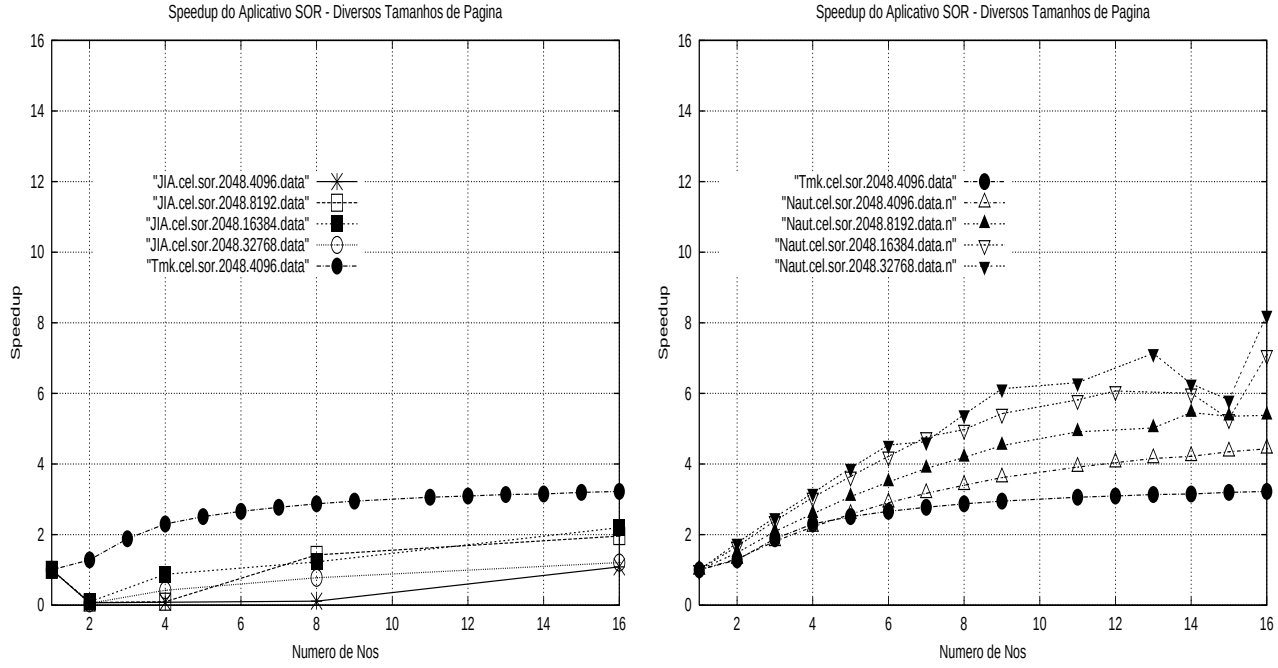

Figura 5.13: *Speedups* do aplicativo SOR (Rice University)- tamanho 2048 - Técnica de Agregação de Páginas

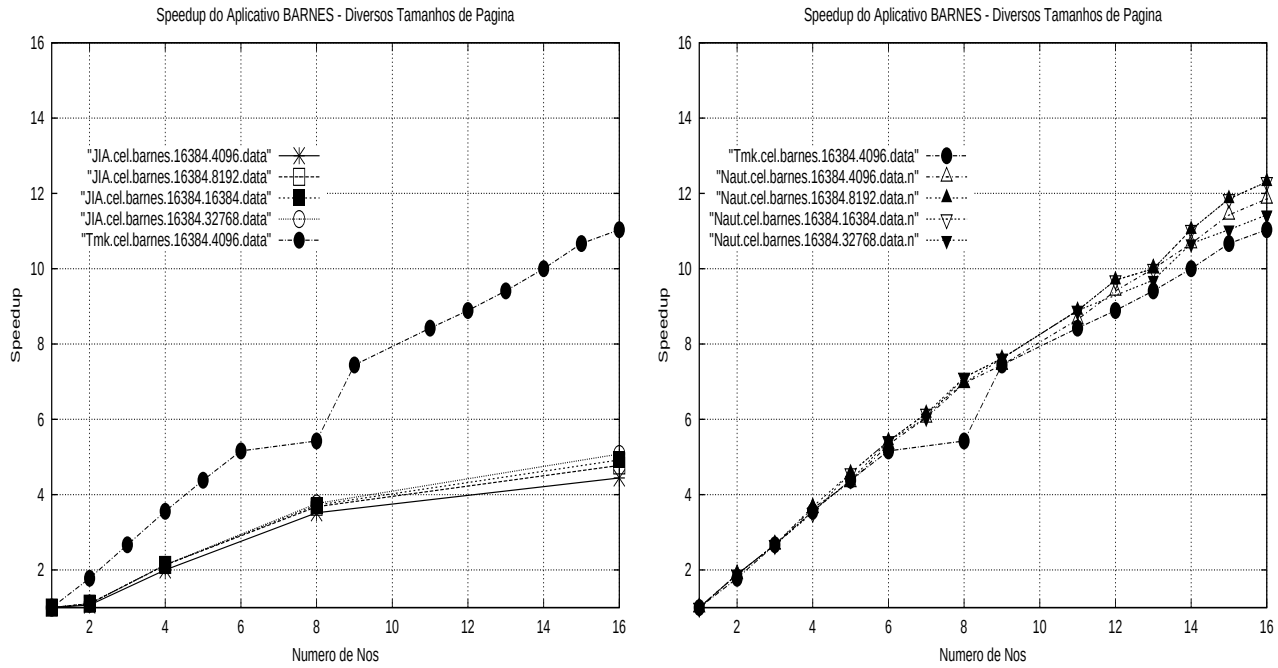

Figura 5.14: *Speedups* do aplicativo Barnes (SPLASH-2) - Técnica de Agregação de Páginas

com o aumento do tamanho de página. Como conseqüência direta da aplicação da agregação, o número de mensagens de páginas foi bastante reduzido, implicando na redução do número global de mensagens. Outro efeito imediato da agregação foi a redução, também alta, do número de SIGSEGVs. Ao contrário dos anteriores, o número de kbytes teve um pequeno aumento devido ao efeito do falso compartilhamento. A grande redução do número de mensagens devido à parcela de páginas acabou sendo responsável pelo aumento do *speedup* do JIAJIA.

Como pode ser visto na figura 5.14 e na tabela 5.3, os *speedups* do Nautilus aumentaram quando se passou do tamanho 4kB para 8kB, mantendo-se de 8kB para 16kB, e diminuindo, quando se passou de 16kB para 32kB. O número de mensagens foi reduzido pela redução do número de mensagens relativo a páginas, quando a técnica de agregação foi aplicada. Também o número de SIGSEGVs foi bastante reduzido, conforme esperado. Ao contrário, houve um aumento do número de kbytes, aumento este causado por efeito do falso compartilhamento. Além disso, o aumento do tamanho da página, acabou acarretando o aumento do tempo de espera por uma página, isto é aumento conseqüente do tempo de empacotamento e desempacotamento, bem como uma provável sobrecarga por pedidos de uma mesma página, causando assim a diminuição de *speedups* para páginas de tamanho maior.

5.7.5 Conclusões Gerais: Técnica de Agregação de Páginas

As conseqüências gerais da aplicação da técnica de agregação de páginas ao JIAJIA e ao Nautilus podem ser resumidas em:

- diminuição do número de mensagens;

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
t(1)	46.73	187.28	750.03	210.22	1148.20	5847.02	75.13	404.20	2063.31	2.86	6.33	11.35	8027.54	320
t(16).Tmk.4096	2.99	11.93	47.76	14.04	74.34	371.88	6.98	33.93	147.28	0.93	1.49	3.52	519.43	29
t(16).JIA.4096	3.13	12.04	47.74	14.83	76.82	380.04	10.31	37.21	169.17	2.18	4.61	10.45	546.01	72
t(16).Naut.4096.n	3.58	14.14	56.47	14.98	76.66	381.3	5.67	30.25	140.36	0.89	1.67	2.57	522.01	27
t(16).Naut.4096.c	-	14.14	56.47	-	80.64	392.00	5.59	28.56	139.69	0.45	0.79	1.08	527.8	27
Sp(16).Tmk.4096	15.67	15.70	15.70	14.97	15.44	15.72	10.76	11.91	14.00	3.07	4.25	3.22	15.45	11.43
Sp(16).JIA.4096	14.95	15.56	15.71	14.18	14.95	15.39	7.29	10.86	12.20	1.31	1.37	1.09	14.70	4.44
Sp(16).Naut.4096.n	13.07	13.24	13.28	14.03	14.97	15.33	13.25	13.36	14.70	3.21	3.79	4.41	15.38	11.85
Sp(16).Naut.4096.c	-	13.24	13.28	-	14.24	14.92	13.45	14.15	14.77	6.35	8.01	10.51	15.21	11.85

Tabela 5.4: Tempos e *speedups* para 16 nós - Técnica de Detecção de Escrita CO-WD

- diminuição do número de páginas que trafegam na rede;
- diminuição do número de SIGSEGVs;
- aumento do *speedup* para os aplicativos MM, SOR e Barnes;
- aumento do número de kBytes para o LU.

5.8 Técnica de Detecção de Escrita CO-WD

Antes de serem exibidas as curvas pertinentes a esta técnica, convém mencionar que não houve nenhuma alteração significativa dos *speedups* (desempenho) no JIAJIA, quando esta técnica foi aplicada e portanto, decidiu-se não exibir os resultados neste caso. Crê-se que existe algum problema na implementação desta técnica para este DSM.

5.8.1 LU (SPLASH-2)

Na figura 5.15 estão plotadas as curvas de *speedup* dos vários DSMs, com o Nautilus utilizando a técnica de detecção de escrita CO-WD.

Neste algoritmo as matrizes são distribuídas entre os nós de forma que cada um deles escreve diretamente em seu nó *home*. Como são utilizadas barreiras entre as fases de computação para sincronizar os nós, no modelo com a aplicação da técnica de detecção tradicional (.n nos gráficos e tabela), todas as páginas compartilhadas são protegidas contra escrita na barreira. O método de detecção de escrita CO-WD(.c nos gráficos e tabela) não escreve nas páginas compartilhadas na barreira, permitindo que as páginas dos nós *home* sejam escritas sem nenhum SIGSEGV (ou SIG nas tabelas).

Como conseqüências da aplicação da técnica de detecção CO-WD, ocorreu uma diminuição do número de mensagens de páginas, de SIGSEGVs e de kbytes para o Nautilus. Porém estas reduções

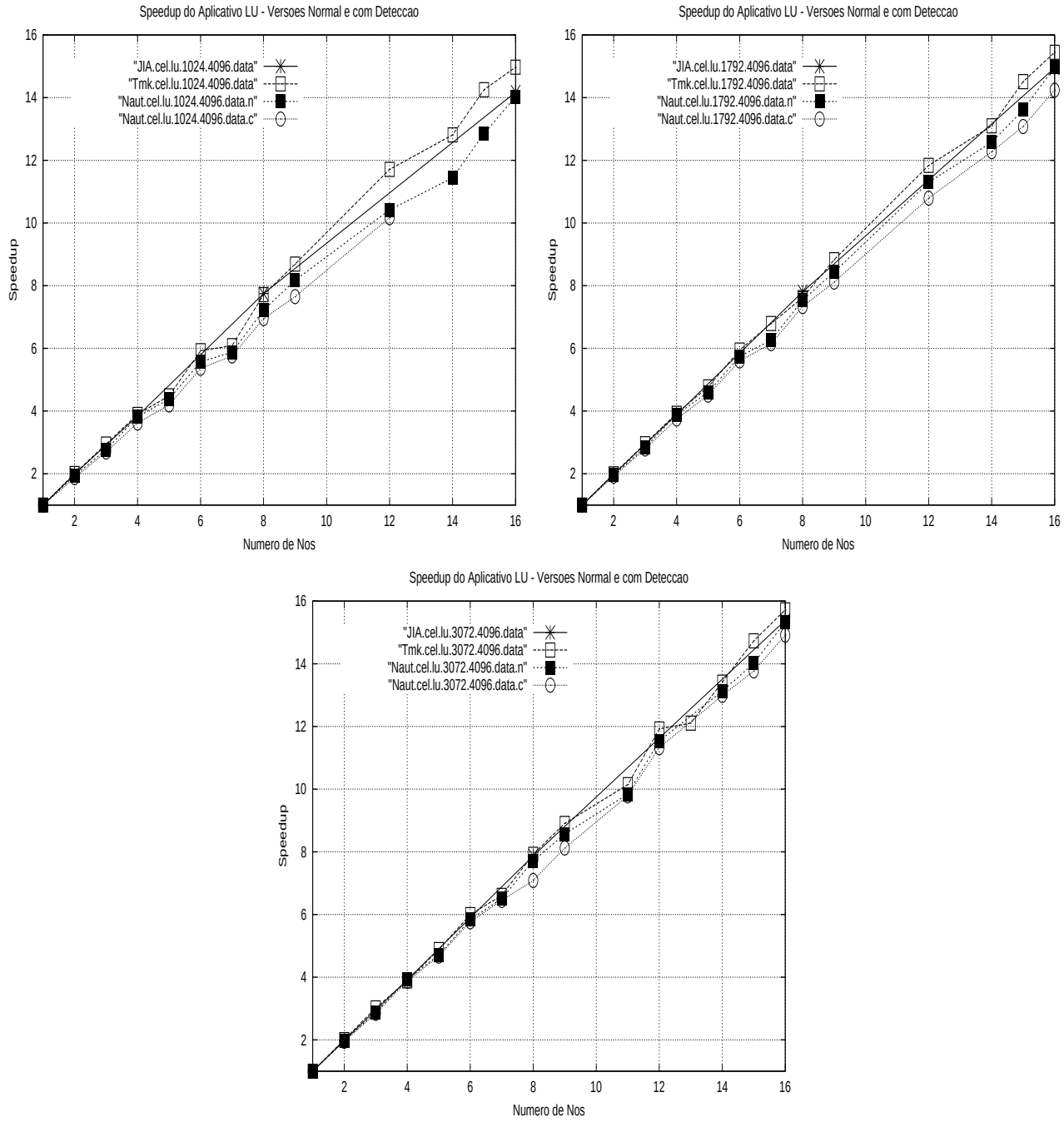

Figura 5.15: *Speedups* do aplicativo LU (SPLASH-2) - tamanhos 1024, 1792 e 2048 - Técnica de Detecção de Escrita CO-WD

não se refletiram no *speedup*. Crê-se que isto acontece devido ao maior *overhead* da implementação de se detectar as escritas somente nas páginas que tem cópias do nó *home*, prejudicando assim o desempenho do Nautilus.

5.8.2 MM

Na figura 5.16 são mostradas as curvas de *speedup* dos vários DSMs, com o Nautilus utilizando a técnica de detecção de escrita CO-WD.

Os *speedups* do Nautilus aumentaram ligeiramente com a aplicação da técnica de detecção CO-WD, como conseqüência da implementação do Nautilus e não da técnica, o que pode ser constatado na figura 5.16 e na tabela 5.4. Isto ocorre porque a técnica de detecção CO-WD somente apresenta resultados ao fim de uma barreira, e como este programa possui uma única localizada ao final, não tendo mais nenhum trecho de programa depois dela, portanto os efeitos não aparecem. O número de mensagens, de kbytes, de páginas e de SIGSEGVs do Nautilus se mantêm.

5.8.3 SOR (Rice University)

Na figura 5.17 são mostradas as curvas de *speedup* do SOR dos vários DSMs, onde o Nautilus utiliza a técnica de detecção de escrita CO-WD.

No aplicativo SOR as matrizes são distribuídas entre os nós de forma que cada um deles escreve diretamente em seu nó *home*. Como são utilizadas barreiras entre as fases de computação para sincronizar os nós, no modelo com a aplicação da técnica de detecção tradicional (.n), todas as páginas compartilhadas são protegidas contra escrita na barreira. O método de detecção de escrita CO-WD não escreve nas páginas compartilhadas na barreira, permitindo que as páginas dos nós *home* sejam escritas sem nenhum SIGSEGV (ou SIG nas tabelas).

A técnica de detecção CO-WD melhorou bastante o desempenho do Nautilus: 97,8% para matriz de tamanho 1024, 111,4% para matriz de tamanho 1536 e 138,3%, para matriz de 2048, resultados que podem ser acompanhados na figura 5.16 e na tabela 5.4. Com reduções no número de mensagens, de kbytes, de mensagens de páginas (que causaram a redução do número de mensagens) e de SIGs, pode-se justificar o grande aumento de desempenho do Nautilus pela não escrita das páginas compartilhadas nos nós *home* em cada barreira, que é o principal objetivo da detecção CO-WD.

5.8.4 Barnes (SPLASH-2)

Na figura 5.18 estão as curvas do aplicativo Barnes para os vários DSMs, onde o Nautilus utiliza a técnica de detecção de escrita CO-WD.

A técnica de detecção CO-WD não trouxe benefícios ao Nautilus para este aplicativo, como se pode notar na tabela 5.4 e na figura 5.18. O número de kbytes aumentou pelo ligeiro aumento

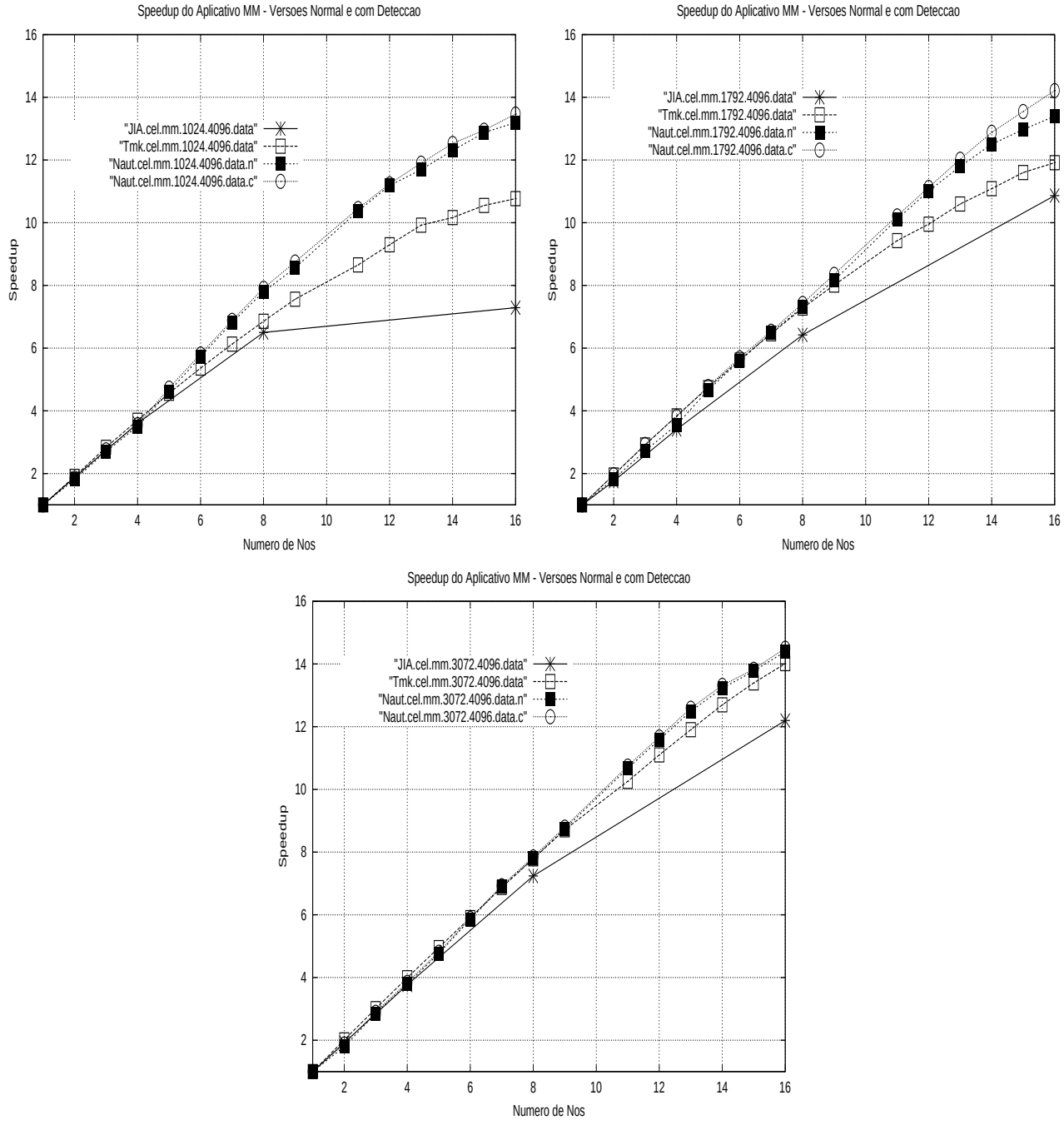

Figura 5.16: *Speedups* do aplicativo MM - tamanhos 1024, 1792 e 3072 - Técnica de Detecção de Escrita CO-WD.

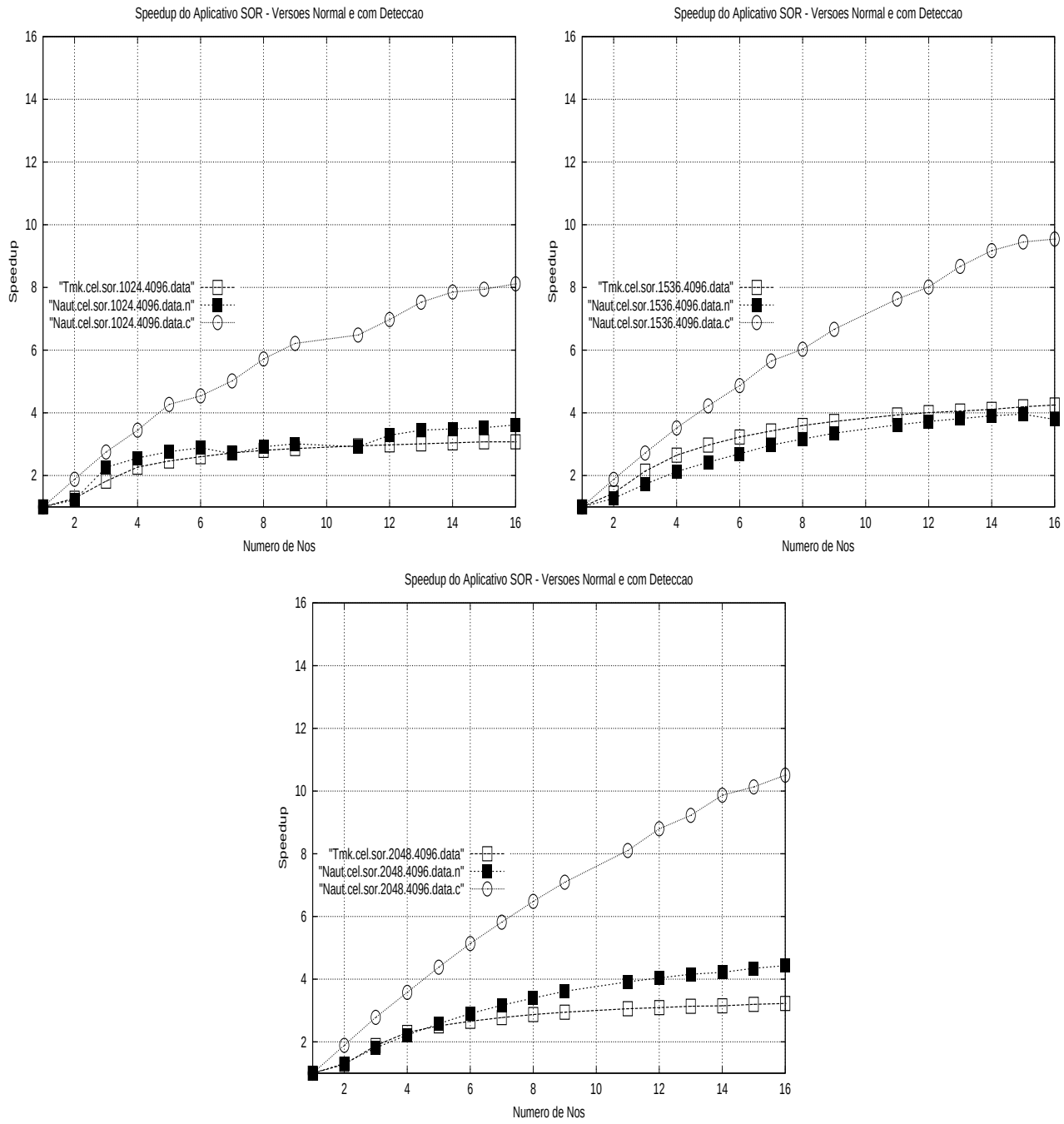

Figura 5.17: *Speedups* do aplicativo SOR (Rice University) - tamanhos 1024, 1536 e 2048 - Técnica de Detecção de Escrita CO-WD

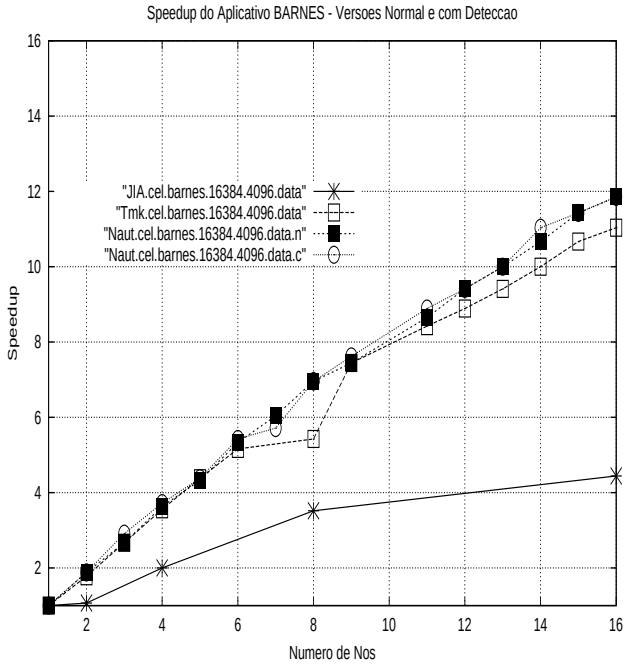

Figura 5.18: *Speedups* do aplicativo Barnes (SPLASH-2) - 16384 corpos - Técnica de Detecção de Escrita CO-WD

do número de mensagens de páginas, que é o pior caso da conseqüência esperada da aplicação da técnica de detecção CO-WD.

5.8.5 Conclusões Gerais: Técnica de Detecção de Escrita CO-WD

Pode-se dizer que com a aplicação da técnica de detecção CO-WD de escrita ao Nautilus, para o aplicativo LU, SOR e Barnes, conseguiu-se uma redução significativa do número de mensagens como efeito da redução do número de mensagens de páginas e do número de SIGSEGVs. Vale salientar que para o SOR houve um grande aumento de *speedup* como conseqüência da aplicação desta técnica pelo fato de não escrever em páginas compartilhadas que estão no nó *home*, não tendo-se a necessidade de transmitir mais *diffs*, isto é permitindo que as páginas compartilhadas das matrizes permanecessem no estado *read-write*.

5.9 Técnica de Agregação de Páginas e Detecção de Escrita CO-WD

Antes de serem exibidas as curvas pertinentes à aplicação conjunta das técnicas de agregação e de detecção CO-WD, convém mencionar que não houve nenhuma alteração significativa dos *speedups* (desempenho) no JIAJIA, quando a técnica de detecção CO-WD foi aplicada e portanto, decidiu-se não exibir os resultados da aplicação das duas técnicas neste caso. Crê-se que existe algum problema na implementação desta última técnica no JIAJIA. Foram incluídos também nas

tabelas os parâmetros referentes ao JIAJIA com a técnica de agregação de página para efeitos comparativos.

Concluindo, somente os *speedups* referentes à versão tradicional (com detecção de escrita tradicional e com página de 4kB) do JIAJIA foram inclusos nos gráficos abaixo por dois motivos: primeiramente, porque a técnica de detecção CO-WD não teve efeito nos tempos de execução; em segundo lugar, para efeitos de melhor compreensão das curvas não foram inclusas, já que conforme dito antes, a técnica de detecção de escrita CO-WD não teve efeito.

Além disso, os fontes do TreadMarks não estavam disponíveis para que novas versões de sua biblioteca com os tamanhos de páginas alterados pudessem ser geradas; assim foi utilizada a versão do TreadMarks com tamanho de página padrão 4kB.

Todas as comparações quantitativas referentes ao Nautilus serão feitas em relação a sua versão básica, isto é, a versão com tamanho de página de 4kB e com a aplicação da técnica de detecção de escrita tradicional.

5.9.1 LU (SPLASH-2)

Na figura 5.19 pode-se visualizar os *speedups* do Nautilus com a aplicação conjunta das técnicas de agregação e detecção CO-WD.

Em relação ao *speedup*, o Nautilus apresenta uma redução com a aplicação de ambas as técnicas, também seguida por consideráveis reduções do número de mensagens e do número de kbytes. Em geral, as percentagens de redução podem ser consideradas como uma 'soma lógica' das reduções apresentadas de cada técnica quando aplicada individualmente. Mesmo com as reduções do número de mensagens (e da parcela mensagens de páginas), do número de kbytes e de SIGSEGVs, o Nautilus apresentou uma redução de *speedup*, que diminuiu com o aumento do tamanho dos parâmetros de entrada. A responsabilidade maior da redução de *speedup* deve-se à técnica de detecção de escrita CO-WD, que possui um maior *overhead* da implementação de se detectar as escritas somente nas páginas que tem cópias do nó *home*, prejudicando assim o desempenho do Nautilus.

5.9.2 MM

Na figura 5.20 estão as curvas de *speedup* para o aplicativo MM submetido aos vários DSMs, avaliando-se a aplicação da agregação e detecção de escrita CO-WD conjuntamente aplicadas ao Nautilus.

Como se pode perceber nesta figura e na tabela 5.5, os *speedups* do Nautilus aumentam de 2,3% e 6,0% para os tamanhos de 1024 e 1792, mantendo-se para o tamanho de 3072. A aplicação conjunta das duas técnicas teve como efeito prático o mesmo resultado da técnica de agregação em relação aos parâmetros que estão sendo observados, já que o programa apresenta somente um ponto de sincronização que o finaliza, não permitindo que a parcela de atuação referente à técnica de detecção CO-WD surte os efeitos desejados. A aplicação conjunta das técnicas praticamente

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
t(1)	46.73	187.28	750.03	210.22	1148.20	5847.02	75.13	404.20	2063.31	2.86	6.33	11.35	8027.54	320
t(16).Tmk.4096	2.99	11.93	47.76	14.04	74.34	371.88	6.98	33.93	147.28	0.93	1.49	3.52	519.43	29
t(16).JIA.4096	3.13	12.04	47.74	14.83	76.82	380.04	10.31	37.21	169.17	2.18	4.61	10.45	546.01	72
t(16).JIA.8192	3.05	11.99	47.88	14.63	76.43	380.70	7.84	32.55	153.67	1.94	2.90	5.79	543.39	67
t(16).JIA.16384	3.09	12.03	47.89	14.66	76.13	380.55	7.01	31.36	149.61	2.41	3.63	5.16	549.85	65
t(16).JIA.32768	3.16	12.01	47.95	14.84	76.21	379.86	6.50	30.99	148.51	3.48	5.64	6.84	568.27	63
t(16).Naut.4096.n	3.58	14.14	56.47	14.98	76.66	381.3	5.67	30.25	140.36	0.89	1.67	2.57	522.01	27
t(16).Naut.8192.n	3.59	14.15	56.47	15.10	76.38	381.1	5.69	30.26	140.37	0.62	1.24	2.11	577.71	26
t(16).Naut.16384.n	3.61	14.16	56.48	15.63	77.07	381.9	5.62	30.25	140.36	0.53	0.94	1.60	539.98	26
t(16).Naut.32768.n	3.65	14.20	56.48	17.50	79.22	383.1	5.56	30.24	140.37	2.51?	2.57	1.38	539.14	28
t(16).Naut.4096.c	-	14.14	56.47	-	80.64	392.00	5.59	28.57	139.69	0.45	0.79	1.08	527.8	27
t(16).Naut.8192.c	3.59	14.14	56.47	17.72	79.90	391.3	5.58	28.56	139.88	0.43	0.78	1.19	525.61	26
t(16).Naut.16384.c	3.61	14.16	56.48	17.87	81.25	386.2	5.57	28.56	140.07	0.50	0.76	1.19	530.64	26
t(16).Naut.32768.c	3.64	14.19	56.48	17.32	81.45	386.5	5.55	28.55	139.89	0.58	0.90	1.17	543.17	28
Sp(16).Tmk.4096	15.67	15.70	15.70	14.97	15.44	15.72	10.76	11.91	14.00	3.07	4.25	3.22	15.45	11.43
Sp(16).JIA.4096	14.95	15.56	15.71	14.18	14.95	15.39	7.29	10.86	12.20	1.31	1.37	1.09	14.70	4.44
Sp(16).JIA.8192	15.33	15.61	15.67	14.37	15.02	15.36	9.58	12.42	13.43	1.47	2.18	1.96	14.77	4.78
Sp(16).JIA.16384	15.15	15.57	15.66	14.34	15.08	15.36	10.72	12.89	13.79	1.18	1.74	2.20	14.60	4.92
Sp(16).JIA.32768	14.79	15.47	15.64	14.17	15.06	15.39	11.56	13.04	13.89	0.82	1.21	1.66	14.13	5.08
Sp(16).Naut.4096.n	13.07	13.24	13.28	14.03	14.97	15.33	13.25	13.36	14.70	3.21	3.79	4.41	15.38	11.85
Sp(16).Naut.8192.n	13.04	13.24	13.28	13.92	15.03	15.34	13.20	13.36	14.70	4.61	5.11	5.37	13.89	12.31
Sp(16).Naut.16384.n	12.97	13.23	13.28	13.45	14.90	15.31	13.37	13.36	14.70	5.40	6.73	7.09	14.87	12.31
Sp(16).Naut.32768.n	12.82	13.19	13.28	12.01	14.49	15.26	13.51	13.37	14.70	1.13	2.46	8.22	14.89	11.43
Sp(16).Naut.4096.c	-	13.24	13.28	-	14.24	14.92	13.45	14.15	14.77	6.35	8.01	10.51	15.21	11.85
Sp(16).Naut.8192.c	13.04	13.24	13.28	11.86	14.37	14.94	13.47	14.15	14.75	6.50	8.12	9.53	15.27	12.31
Sp(16).Naut.16384.c	12.95	13.23	13.28	11.76	14.13	15.14	13.48	14.15	14.73	5.72	8.33	9.53	15.13	12.31
Sp(16).Naut.32768.c	12.86	13.19	13.28	12.13	14.10	15.13	13.55	14.16	14.75	4.93	7.03	9.70	14.78	11.43

Tabela 5.5: Tempos e *speedups* para 16 nós - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

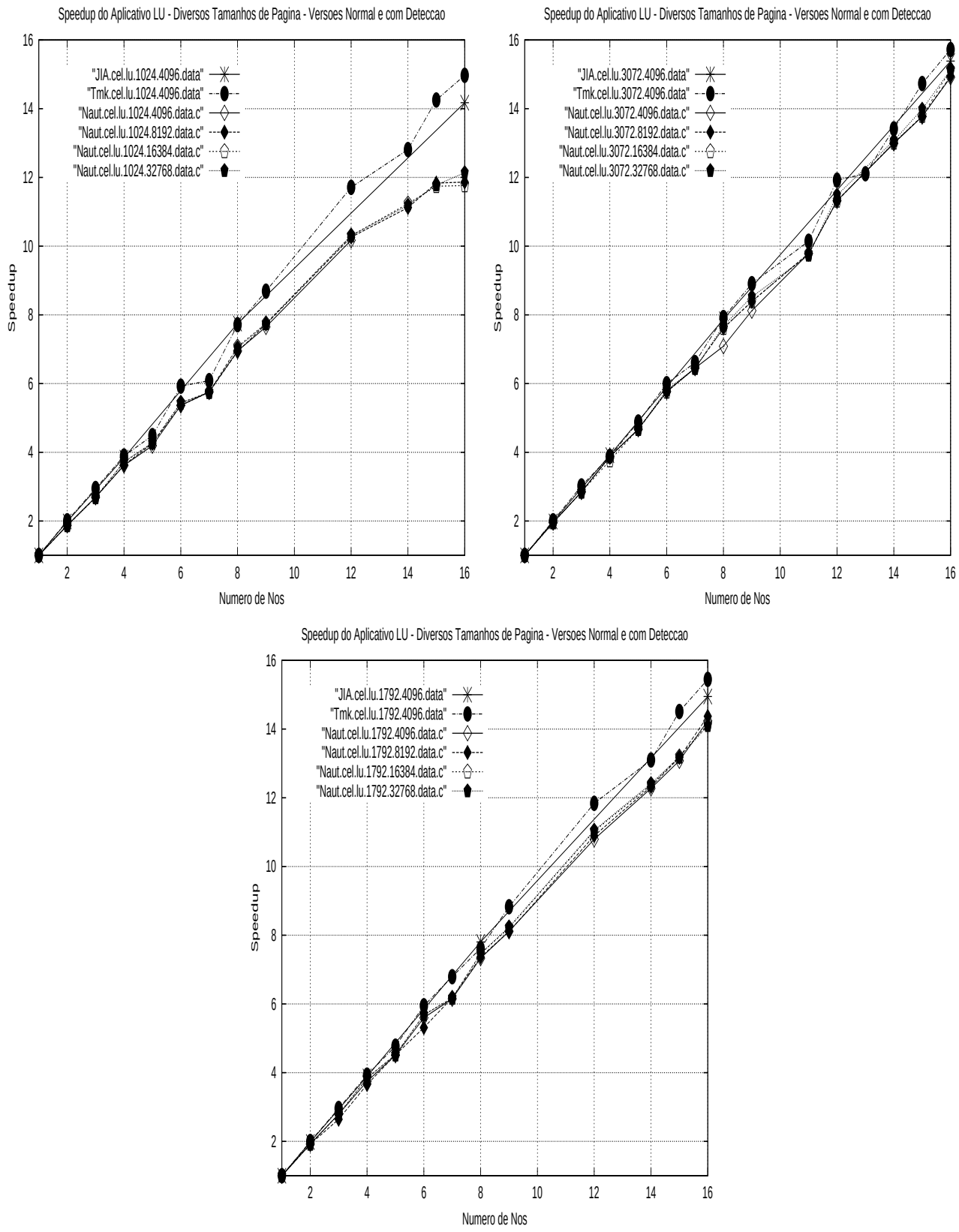

Figura 5.19: *Speedups* do aplicativo LU (SPLASH-2) - tamanhos 1024, 1792 e 3072 - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

não apresentou efeitos sobre os parâmetros número de mensagens, de kbytes, de mensagens de páginas, de SIGSEGVs e de mensagens de *diffs*, seguindo o comportamento geral obtido somente pela aplicação da agregação.

5.9.3 SOR (Rice University)

Na figura 5.21 podem ser vistos os *speedups* do aplicativo SOR submetido aos vários DSMs, onde está sendo avaliada a aplicação de conjunta de ambas as técnicas no Nautilus.

Com a aplicação das duas técnicas, ocorreu um aumento de *speedup* no Nautilus maior do que com a aplicação individual de cada uma delas, portanto valendo a mesma idéia de 'somar' os efeitos de cada uma delas, isto é, reduções maiores de número de mensagens, de mensagens de páginas, de kbytes e de SIGSEGVs.

A aplicação conjunta das duas técnicas permitiu agregar o que cada uma delas tem de melhor, resultando um aumento de *speedup*.

5.9.4 Barnes (SPLASH-2)

Na figura 5.22 estão os *speedups* do aplicativo Barnes, submetido aos vários DSMs, onde estão sendo avaliadas ambas as técnicas no DSM Nautilus.

Como resultado conjunto da aplicação das duas técnicas ao Nautilus, foi obtido o mesmo resultado da aplicação da agregação, como se pôde ver anteriormente na análise individual desta técnica: redução do número de mensagens, de mensagens de páginas e de SIGSEGVs, acompanhados pelo aumento do número de kbytes. Estes resultados permitiram um ligeiro aumento de *speedup* do Nautilus.

5.9.5 Conclusões Gerais: Técnica de Agregação de Páginas e Detecção de Escrita CO-WD

Pode-se considerar o resultado da aplicação conjunta das técnicas de agregação e detecção CO-WD como sendo um 'OR' da aplicação das duas técnicas, isto é, reduções (incrementos) individuais das duas técnicas resultam numa redução (incremento) maior na aplicação conjunta.

5.10 Conclusão Geral da Aplicação das Técnicas Estudadas

Neste capítulo pôde-se comparar diretamente três sistemas DSM, TreadMarks, JIAJIA e Nautilus, confrontando-os em vários aspectos e para vários aplicativos, verificando qual deles apresentam o melhor *speedup* para uma série de aplicativos.

A aplicação da técnica de agregação permitiu um aumento de *speedup* para os programas MM, SOR e Barnes para os DSMs JIAJIA e Nautilus, onde a técnica foi aplicada.

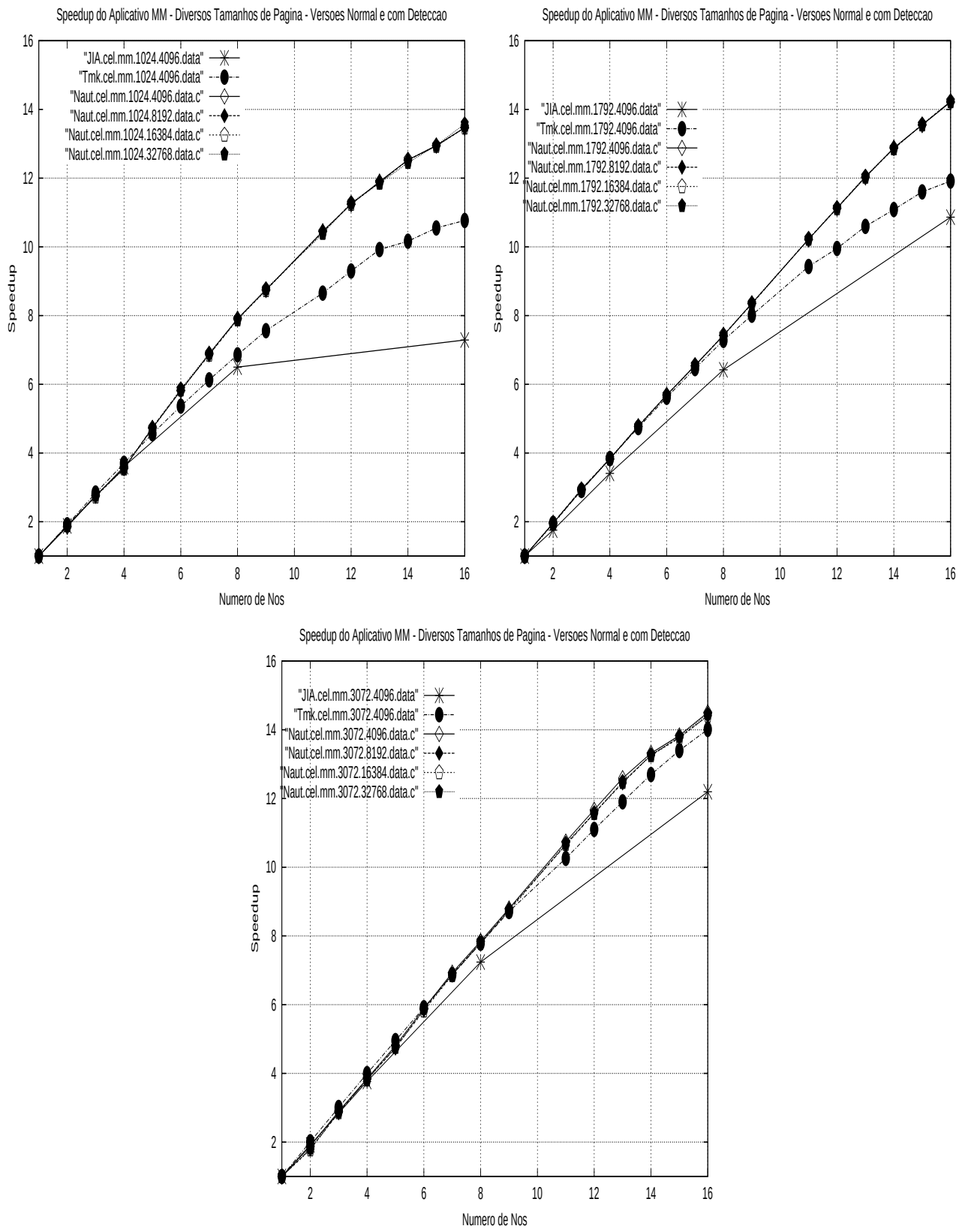

Figura 5.20: *Speedups* do aplicativo MM - tamanhos 1024, 1792 e 3072 - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

Como conclusão geral, o tamanho de página obtido para o JIAJIA que permitiu os melhores *speedups* foram de 16kB e 32kB. Para o Nautilus, os tamanhos mais adequados para obter um aumento do *speedup* foram de 8kB e 16kB.

Uma futura intenção é permitir que o sistema avalie de maneira automática os diferentes tamanhos de página, e monte as tabelas de avaliação, entre elas, as de tempos e *speedups*. Com esta tabela, automaticamente o sistema vai saber quais os melhores *speedups* e decidir o tamanho de página mais adequado. Desta forma, o parâmetro tamanho de página poderia ser modificado de maneira adaptativa a cada *benchmark* escolhido.

O único aplicativo que se beneficiou da aplicação da técnica de detecção de escrita CO-WD foi o SOR. Neste caso houve grande melhora de desempenho pela não escrita das páginas compartilhadas nos nós *home* nas barreiras. De maneira análoga à anterior, uma possível possibilidade é o sistema aplicar esta técnica montando automaticamente as tabelas de avaliação, decidindo se a aplicação vale ou não a pena através de uma consulta da tabela de *speedups*. Escolhido se vale ou não, o sistema poderia se adaptar a cada aplicativo submetido.

Como se pôde perceber, não há uma solução comportamental ótima para todos os casos, sendo necessário estudar caso a caso qual técnica que melhor se adeqúe para a obtenção do melhor desempenho.

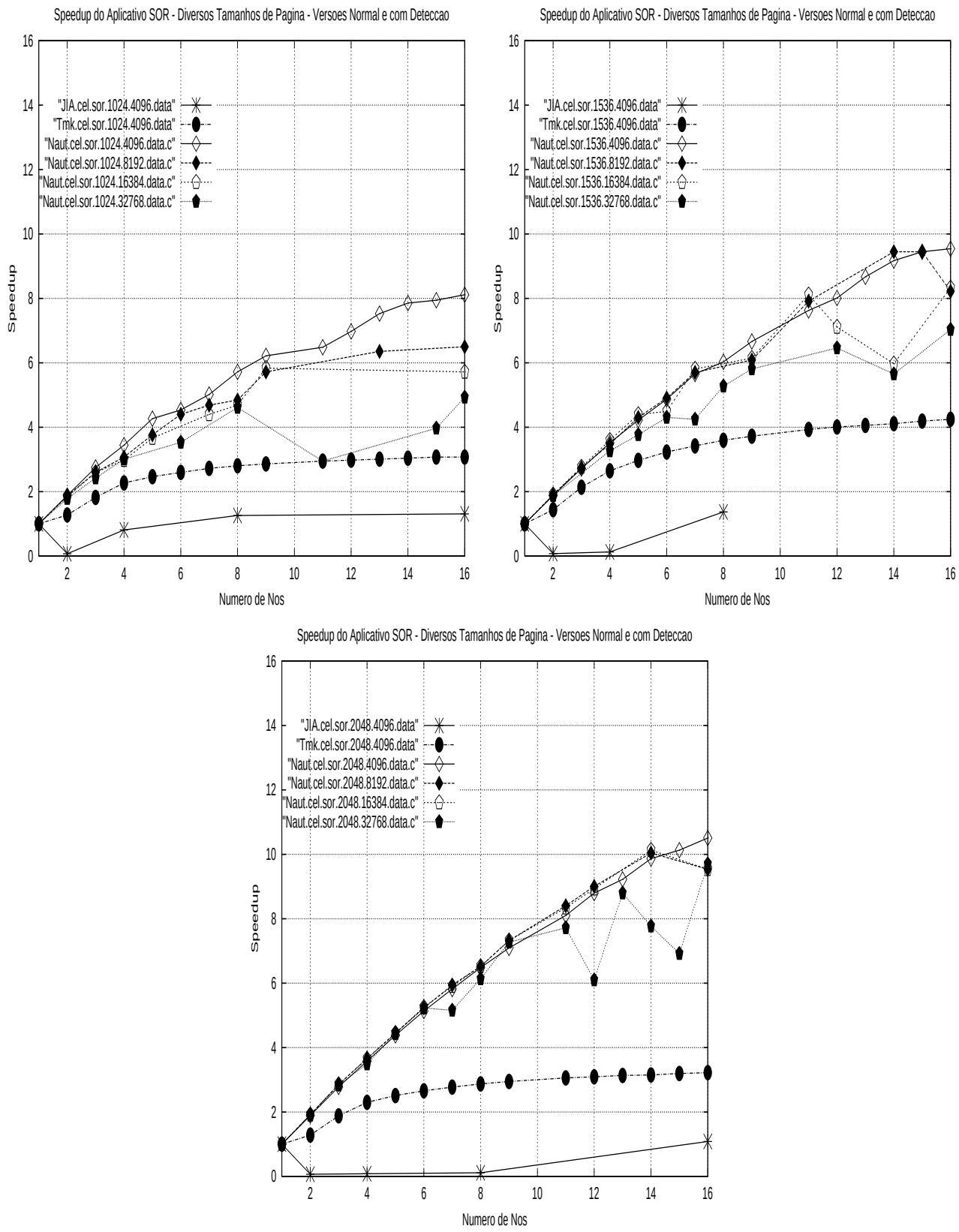

Figura 5.21: *Speedups* do aplicativo SOR - tamanhos 1024, 1536 e 2048 - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

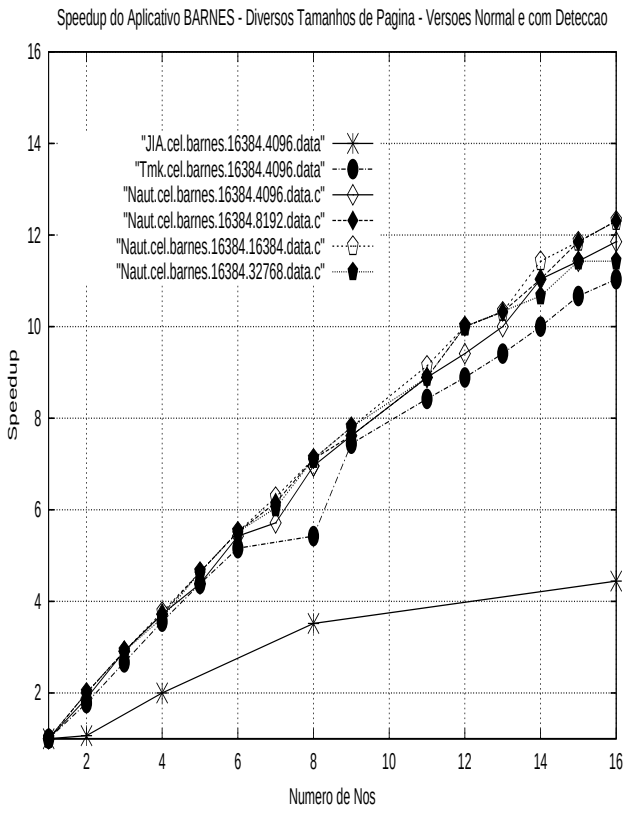

Figura 5.22: *Speedups* do aplicativo Barnes (SPLASH-2)- 16384 corpos - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

Capítulo 6

Discussão de Propostas de Trabalho e Conclusões

Este capítulo tem como objetivo principal esclarecer ao leitor os aspectos não abordados neste trabalho e as futuras atividades que se pretendem seguir.

Além disso, neste capítulo serão apresentadas as conclusões aqui adquiridas.

6.1 Aspectos Não Abordados e Trabalhos Futuros

Este trabalho, como qualquer trabalho em pesquisa não abordou uma série de aspectos que mereceriam ser isoladamente tratados como futuros outros aspectos de pesquisa. Entre eles podem ser citados:

- adquirir uma versão completa do TreadMarks com os fontes disponíveis; desta forma pode-se ter com alguns experimentos mais conclusões a respeito da técnica de agregação de páginas;
- executar a mesma avaliação feita neste trabalho com outros DSMs;
- aumento dos parâmetros de entrada dos *benchmarks* para abranger uma faixa maior em que se possam garantir de maneira ainda mais ampla os resultados obtidos; para isto, é necessário o aumento da memória física disponível no *cluster* de teste, de atualmente 128MB para 1,028 GB;
- executar a mesma avaliação de desempenho para outros tipos de redes como a GigaEthernet, SCI, Myrinet e ATM; desta forma, os DSMs vão poder ser avaliados em outras redes e também o aumento do tamanho das mensagens, que é conseqűente da técnica de agregação poderá melhor ser investigado;
- portar o Nautilus para máquinas multiprocessadas com o intuito de avaliar seu comportamento aproveitando a memória compartilhada física para manter a coerência e aproveitando

o fato de ter sido feito com o pacote *pthreads*, que permite facilmente a utilização de vários processadores;

- modificar a implementação das barreiras pela utilização de primitiva de *multicast*, contra a atual de *broadcast*;
- implementar a técnica de *home-migration* [21] para o Nautilus;
- efetuar a integração do Nautilus com pacotes de comunicação a nível de usuário como o VIA, de forma a averiguar os efeitos da minimização dos *overheads* dos protocolos de comunicação;
- efetuar a integração do Nautilus com uma linguagem de programação paralela que permita maior facilidade de programação ao usuário; esta linguagem seria acompanhada por um compilador que facilitaria a geração de código. Para isto, tem-se o intuito de utilizar a linguagem/compilador CPAR[63](que pode gerar código para vários DSMs) de modo a facilitar a avaliação e difusão dos DSMs pela comunidade acadêmica;
- procurar outros *benchmarks* como programas que lidem com elementos finitos, cálculos relacionados ao DNA e proteínas, bastante em moda nestes dias;
- modificar o Nautilus de forma que ao invés de se ter coerência de áreas de memória, ter coerência de áreas em disco; assim se poderia ter um sistema de discos coerente;
- implementar uma forma automática para que o Nautilus monte as tabelas de avaliação para cada técnica aplicada, agregação e detecção CO-WD, e através da análise destas tabelas, entre elas as de tempo e *speedup*, o melhor *speedup* para cada aplicativo seria escolhido; a escolha seria baseada na execução de cada aplicativo com cada uma das técnicas individualmente (ou em conjunto) aplicada (s) e montagem da tabela; após o término da tabela, o sistema poderia incorporar os parâmetros mais adequados a cada aplicativo;
- estudo de técnicas de reestruturação de programas aplicativos que possam melhorar seu desempenho[27, 28];
- aplicar as principais idéias de modelos de consistência a áreas como banco de dados e coerência de *cache* em servidores *web*, pois são áreas com características muito semelhantes.

6.2 Conclusões sobre a Pesquisa

Neste ponto citam-se as principais conclusões da pesquisa realizada neste trabalho:

- foi realizada uma revisão bibliográfica sobre os vários DSMs, com suas principais características e delas decorrentes, suas vantagens e desvantagens;

- foi proposto um novo DSM, compatível em termos de programas aplicativos e com características diferentes dos demais;
- foi estabelecido de forma prática uma metodologia que engloba um ambiente e alguns parâmetros onde se pode concluir a respeito do desempenho de sistemas DSM, em outras palavras é uma iniciativa de criar um laboratório para avaliação comparativa entre DSMs;
- foi implementada uma análise quantitativa bastante detalhada sobre os parâmetros envolvidos nas comparações entre os DSMs;
- foi proposta uma nova classificação de sistemas DSM baseada numa extensão da classificação de Carter[10], que é mais adequada porque engloba os modelos de consistência mais modernos;
- vários artigos foram publicados como resultados intermediários deste trabalho.

6.3 Conclusões sobre os Resultados

Neste item citam-se as principais conclusões sobre os resultados aqui obtidos:

- implementou-se o DSM que foi proposto: o Nautilus; este foi avaliado juntamente com outros DSMs amplamente utilizados pela comunidade acadêmica, o TreadMarks e o JIAJIA[39, 40, 41, 42, 43, 44];
- com a implementação operando, o Nautilus passou a ser o primeiro DSM baseado em páginas a não utilizar o sinal SIGIO[41];
- o Nautilus é o primeiro DSM onde foram aplicadas simultaneamente as técnicas de agregação de páginas e detecção de escrita CO-WD[46];
- a técnica de agregação permitiu aumentos de *speedup* nos aplicativos Barnes, SOR e MM; estes aumentos de *speedup* foram acompanhados pela diminuição do número de mensagens em decorrência da diminuição do número de páginas, e pela diminuição do número de SIGSEGVs, portanto do número de vezes que o respectivo *handler* foi acionado;
- os tamanhos de página escolhidos como resultado de melhores *speedups* pela aplicação da técnica de agregação foram 8kB e 16kB;
- somente o aplicativo SOR apresentou melhora em seu *speedup* com a aplicação da técnica de detecção de escrita CO-WD aplicada ao Nautilus devido à minimização das escritas nas páginas compartilhadas nas barreiras;

- pode-se considerar o resultado da aplicação conjunta das técnicas de agregação e detecção CO-WD como sendo um 'OR' da aplicação das duas técnicas, isto é, reduções (incrementos) individuais das duas técnicas resultam numa redução (incremento) maior na aplicação conjunta;

- não existe uma solução ótima que satisfaça a maioria dos casos, ou mesmo todos eles, sendo necessário estudar-se caso a caso, aplicando as técnicas e descobrindo qual delas melhor se aplica.

Referências Bibliográficas

[1] *Accelerating the Standard for Speed.* Disponível em <http://www.gigabit-ethernet.org/index.html>. Acesso em outubro de 1999.

[2] *Active Messages.* Disponível em <http://now.cs.berkeley.edu/AM/active_messages.html>. Acesso em abril de 1998.

[3] *Adsmith DSM.* Disponível em <http://archi1.ee.ntu.edu.tw/~wyliang/adsmith>. Acesso em julho de 1998.

[4] *AMD Athlon Processor.* Estados Unidos da América. Disponível em <http://www.amd.com>. Acesso em dezembro de 1999.

[5] AMZA C.; COX A. L.; SWARKADAS S. JIN L. J.; RAMAJANI K.; ZWAENEPOEL W. Adaptive Protocols for Software Distributed Shared Memory. Artigo em revista. Proceedings of IEEE, Special Issue on Distributed Shared Memory, pp. 467-475, março, 1999.

[6] BERSHAD B. N.; ZEKAUSKAS M. J.; SAWDON W. A. The Midway Distributed Shared Memory System. Artigo de revista. COMPCOM, 1993.

[7] BUONADONNA P. *Unet Berkeley Project.* Universidade da Califórnia, Berkeley, Estados Unidos da América. Disponível em <http://www2.cs.cornell.edu/U-Net/Default.html>. Acesso em junho de 1998.

[8] *BUONADONNA P. Berkeley VIA Project, An Investigation of the Virtual Interface Architecture.* Universidade da Califórnia, Berkeley, Estados Unidos da América, 1998. Disponível em <http://www.cs.berkeley.edu/~philipb/via>. Acesso em setembro de 1998.

[9] CARTER J. B.; KHANDEKAR D.; KAMB L. *Distributed Shared Memory: Where We are and Where we Should Headed.* Technical Report. Computer Systems Laboratory, Universidade de Utah, 1995.

[10] CARTER J. B.; *Efficient Distributed Shared Memory Based on Multi-protocol Release Consistency.* Tese de doutorado. Universidade de Rice , Houston, Texas, setembro, 1993.

[11] CARTER J. B.; BENNETT J. K., ZWAENEPOEL W. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory System. Artigo de Revista. ACM Transactions on Computer Systems, Vol. 13, no. 3, pp. 205-244, agosto, 1995.

[12] *Cashemere Project*. Universidade de Rochester. Nova Iorque. Estados Unidos da América. Disponível em <http://www.rochester.edu/research/cashemere>. Acesso em outubro de 1999.

[13] CIACCIO G. *GAMMA Project: Genoa Active Message MAchine*. Disponível em <http://www.disi.unige.it/project/gamma/>. Acesso em fevereiro de 1998.

[14] COX A.L.; LARA E.; HU Y.C.; ZWAENEPOEL W. *A Performance Comparison of Homeless and Home-based Lazy Release Consistency Protocols in Software Shared Memory*. Artigo em evento. Proceedings of the Fifth High Performance Computer Architecture Conference (HPCAC), janeiro, 1999.

[15] *Distributed Inter-Process Communication (DIPC)* System. Disponível em <http://wallybox.cei.net/dipc>. Acesso em maio de 1999.

[16] ESKICIOGLU M. R.; MARSLAND T. A.; HU W.; SHI W. *Evaluation of JIAJIA Software DSM System on High Performance Architectures*. Artigo de evento. Thirty-Second Hawaii International Conference on System Sciences(HICSS-32), Havaí, Estados Unidos da América, janeiro, 1999.

[17] *Fast-messages* package. Estados Unidos da América. Disponível em <http://www_csag.uscd.edu/projects/comm/fm.html>. Acesso em setembro de 1999.

[18] HU W.; SHI W.; TANGZ. *Adaptive Write Detection in Home-based Software DSMs*. Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, Redondo Beach, Califórnia, Estados Unidos da América, agosto, 1999.

[19] HU W.; SHI W.; TANG Z.; LI M., A Lock-based Cache Coherence for Scope Consistency. Artigo em revista. Journal of Computer Science and Technology, Vol 13, No. 2, pp. 97-110, 1998.

[20] HU W.; SHI W.; TANG Z. *JIAJIA: An SVM System Based on A New Cache Coherence Protocol*. Artigo em evento. Proceedings of the High Performance Computing and Networking (HPCN'99), LNCS 1593, pp. 463-472, Springer, Amsterdam, Países Baixos, abril, 1999.

[21] HU W.; SHI W.; TANG Z. *Home Migration in Home-Based Software DSMs*. Artigo em evento. 1st Workshop on Software Distributed Shared Memory (WSDSM '99), Rhodes, Grécia, 25 de junho, 1999.

[22] HU W.; SHI W.; TANG Z. *Reducing System Overheads in Home-based Software DSMs.* Artigo em evento. 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing(IPPS/SPDP'99), IEEE CS Press, pp. 167-173, San Juan, Puerto Rico, abril, 1999.

[23] HU W; SHI W.; TANG Z. *Write Detection in Home-based Software DSMs.* Artigo em evento. Proceedings of Euro-Par'99, , Tolouse, France, janeiro, August 31-September 2, 1999.

[24] IFTODE L. *Home-based Shared Virtual Memory.* Tese de doutorado. Universidade de Princeton , Estados Unidos da América, agosto, 1998.

[25] IFTODE L; SINGH J. P.; Iftode L., Singh J. P. Shared Virtual Memory: Progress and Challenges. Artigo de revista. Proceedings of the IEEE, Vol 87, No. 3, março, 1999.

[26] IFTODE L.; SINGH J. P.; LI K. *Scope Consistency: A bridge between release consistency and entry consistency.* Artigo em evento. Proceedings of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA'96), pp. 277-287, junho, 1996.

[27] IFTODE L.; SINGH J.P.; LI K. *Understanding Application Performance on Shared Virtual Memory.* Artigo de evento. Proceedings of the 23rd Annual International Symposium on Computer Architecture, maio, 1996.

[28] JIANG D.; SHAN H.; SINGH J. P. *Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors.* Artigo em evento. Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming, junho, 1997.

[29] JOHNSON D. B.; ZWAENEPOEL W. The Peregrine High-Performance RPC System. Artigo em revista. Software: Practice and Experience, Vol. 23, No. 2, pp. 201-221, fevereiro, 1993.

[30] KARLSSON S.;BRORSSON M. *An Infrastructure for Portable and Efficient Software DSM.* Artigo em evento. 1st Workshop on Software Distributed Shared Memory (WSDSM '99), Rhodes, Grécia, 25 de junho, 1999.

[31] KELEHER P. *Distributed Shared Memory Home Pages.* Estados Unidos. Universidade de Maryland. Disponível em <http://www.cs.umd.edu/~keleher/dsm.html>. Acesso em setembro de 1997.

[32] KELEHER P. *Lazy Release Consistency for Distributed Shared Memory.* Tese de doutorado. Universidade de Rochester, Texas, Houston, janeiro, 1995.

[33] KELEHER P. *The Relative Importance of Concurrent Writers and Weak Consistency Models.* Artigo de evento. Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16), pp. 91-98, maio, 1996.

[34] KELEHER P. Update Protocols and Cluster-based Shared Memory. Artigo de revista. In Computer Communications, 22(11), pp. 1045-1055, julho, 1999.

[35] KELEHER P. *Update Protocols and Iterative Scientific Applications.* Artigo de evento. The 12th International Parallel Processing Symposium, março, 1998.

[36] *Larchant Project.* INRIA. França. Disponível em <http://www-sor.inria.fr/SOR/projects/larchant.html>. Acesso em setembro de 1999.

[37] LI K. *Shared Virtual Memory on Loosely Coupled Multiprocessors*, tese de doutorado, Yale University, 1986.

[38] LU H.; DWARKADAS S.; COX A. L.; ZWAENEPOEL W. *Quantifying the Performance Differences between PVM and TreadMarks.* Artigo em revista. Journal of Parallel and Distributed Computation, Vol. 43, No. 2, pp. 65-78, junho, 1997.

[39] MARINO M. D.; CAMPOS G.L. *A DSM Speedup Comparison: TreadMarks, JIAJIA and Nautilus.* Artigo em evento. Proceedings of the International Conference on Parallell and Distributed Processing Techniques and Applications (PDPTA), Vol. IV, pp. 1744-1748, Las Vegas, Nevada, Estados Unidos da América, julho, 1999.

[40] MARINO M. D.; CAMPOS G. L. *An Evaluation of Page Aggregation Technique on Different DSM Systems.* Third International Symposium on High Performance Computing (ISHPC2K), LLNCS 1940, Springer Verlag, v1, pp134-145, Tokio, Japão, outubro, 2000.

[41] MARINO M.D.; CAMPOS G. L.; SATO L.M. *An Evaluation of the Speedup of Nautilus DSM System*, Parallel Symposyum(IASTED PDCS), Boston, proceedings em CD, novembro, 1999.

[42] MARINO M. D.;, CAMPOS G.L. *Nautilus: A Three Third Generation DSM System*, 8th Workshop on Shared Memory Multiprocessors in conjunction with ISCA99, Atlanta, Georgia, Estados Unidos da América, abril/maio, 1999.

[43] MARINO M. D.; CAMPOS G.L. *A Preliminary Comparison Between Two Scope Consistency DSM Systems: JIAJIA and Nautilus.* Artigo em evento. Proceedings of the International Workshop on Parallel Computing at Seventh International Conference on Parallel Processing., IEEE proceedings, v1., pp319-324, Fukujima, Japão, julho, 1999.

[44] MARINO M.D.; CAMPOS G. L. A Preliminary DSM Speedup Comparison: JIAJIA x-Nautilus. Capítulo de livro. Kluwer Academics, HPCS99, Kingston, Canada, junho, 1999.

[45] MARINO M. D.; CAMPOS G. L. *A Preliminary Study of Cache-Only Write Detection Technique for Nautilus DSM.* Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho (SBAC-PAD), v1, pp 217-224, Águas de São Pedro, São Paulo, Brasil, outubro, 2000.

[46] MARINO M. D.; CAMPOS G.L. *Techniques for Improving the Speedup of Nautilus DSM.* Artigo em evento. Proceedings of the International Conference on Parallell and Distributed Processing Techniques and Applications (PDPTA), proceedings em CD, CSREA, Las Vegas, Estados Unidos da América, junho, 2000.

[47] MARINO M. D.; CAMPOS G. L. *The Speedup of Nautilus, a new DSM System, Compared to Treadmarks.* Second International Conference on Parallel and Computing Systems (PCS99), v1. p70-78, Baja California, México, agosto, 1999.

[48] MARINO, M.D. *Pulsar: Um Sistema de Memória Distribuída e Compartilhada baseado em Consistência de Memória Relaxada*, Dissertação de Mestrado, Escola Politécnica da Universidade de São Paulo, outubro, 1996.

[49] *Mirage Project.* Disponível em <http://cs.ucr.edu/projects/mirage.html>. Acesso em novembro de 1999.

[50] MONNERAT L.R.; BIANCHINI R. *Efficiently Adapting to Sharing Patterns in Software DSMs.* Artigo em evento. Proceedings of the 4th IEEE International Symposium on High-Performance Computer Architecture (HPCA98), fevereiro, 1998.

[51] *Mosix Scalable Computing for Linux.* Instituto de Ciências da Computação, Universidade de Hebrew, Jerusalem, Israel. Disponível em <http://www.mosix.cs.huji.ac.il>. Acesso em maio de 1999.

[52] MOWRY T. C.; Cnah Q. C.; LO A. K. W. *Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory.* Artigo em evento. Proceedings of the 4th IEEE Symposium on High Performance Computer Architechture, 1998.

[53] *MPICH - A Portable Implementation of MPI.* Laboratório Nacional de Argone. Estados Unidos da América. Disponível em <http://www-unix.mcs.anl.gov/mpi/mpich>. Acesso em agosto de 1999.

[54] *Myrinet Overview.* Disponível em <http://www.myri.com>. Acesso em novembro de 1997.

[55] NG M.C.; WONG W.F. *Adaptive schemes for home-based DSM systems.* Artigo em evento. 1st Workshop on Software Distributed Shared Memory (WSDSM '99) Rhodes, Grécia, 25 de junho, 1999.

[56] N. P.; MANOJ R. *CAS-DSM: A compiler-assisted DSM.* Artigo em evento. 1st Workshop on Software Distributed Shared Memory (WSDSM '99) Rhodes, Grécia, 25 de junho, 1999.

[57] *OpenMP Resources.* Disponível em <www.openmp.org/index.cgi?resources>. Acesso em novembro de 1999.

[58] *Parallel I/O Archive* (PARIO). Faculdade Dartmouth, Departamento de Computação. Estados Unidos da América. Disponível em <http://www.cs.dartmouth.edu/pario>. Acesso em agosto de 1999.

[59] *Parmon*. Disponível em <http://www.dgs.monash.edu.au/~rajkumar/papers/parmon.html>. Acesso em novembro de 1998.

[60] *PVM - Parallel Virtual Machine*. Laboratório Nacional de Oak Ridge. Estados Unidos da América. Disponível em <http://www.epm.ornl.gov/pvm>. Acesso em agosto de 1999.

[61] RAJKMAR B. *High Performance Cluster Computing: Arquitectures and Systems*. Austrália. Disponível em <http://www.dgs.monash.edu.au/~rajkumar/cluster/index.html>. Acesso em novembro de 1998.

[62] SAMANTA R.; BILAS A.; IFTODE L.; SINGH J. P. *Home-based SVM protocols for SMP clusters: Design and Performance*. Artigo de evento. In Proceedings of the 4th International Symposium on High-Performance Computer Architecture, fevereiro, 1998.

[63] SATO L. M.; MIDORIKAWA E. T.; BERNAL V. B. *Práticas em Programação Paralela*. Tutorial. Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho (SBAC-PAD), v1, pp. 1-38, São Paulo, 1992.

[64] SCALES D. J.; GHARACHORLOO K.; THEKKATH C. *Shasta: A Low Overhead, Software Only Approach for Supporting Fine-Grain Shared Memory*, Artigo em evento. Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 297-306, outubro, 1996.

[65] *Shared Virtual Memory Library (SVMlib)*. Disponível em <http://www.lfbs.rwth-aachen.de/~sven/SVMlib>. Acesso em janeiro de 1999.

[66] SHI W.; HU W.; TANG Z. An Interaction of Cache Coherence Protocol and Memory - Consistency Model. Artigo em revista. ACM Operating Systems Review, Vol. 31, No. 4, pp. 41-54, outubro, 1997.

[67] SHI W.; MAO Y.; TANG Z. *Communication Substrate for Software DSMs*. Artigo em evento. Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems, , MIT, Boston, Estados Unidos da América, novembro (3-6), 1999

[68] SHI W., MA J.; TANG Z. *High Efficient Parallel Computation of Resonant Frequencies of Waveguided Loaded Cavetives on JIAJIA Software DSM System*. Artigo em evento. In Proceedings of the High Performance Computing and Networking (HPCN'99), LNCS 1593, pp. 1147-1150, Springer, Amsterdam, Países Baixos, abril, 1999.

[69] SHI W.; HU W.; TANG Z. Where Does the Time Go in SVM System?-Experiences with JIAJIA. Artigo em revista. Journal of Computer Science and Technology, Vol 14, No.3, pp.193-205, maio, 1999.

[70] *Shrimp Princeton Project.* Universidade de Princeton. New Jersey. Estados Unidos da América. Disponível em <http:/www.cs.Princeton.EDU:80/shrimp>. Acesso em julho de 1999.

[71] *Simple Coma.* Disponível em <http://playground.Sun.COM:80/pub/S3.mp/simple-coma/>. Acesso em abril de 1999.

[72] SPEIGHT E.; BENNETT J. K. *Brazos: A third generation DSM system.* Artigo em evento. Proceedings of the 1997 USENIX Windows/NT Workshop, pp. 95-106, agosto, 1997.

[73] SPEIGHT E.; BENNETT J. K. *Using Multicast and Multithreading to Reduce Communication in Software DSM Systems,*. Artigo em evento. Proceedings of the Fourth Symposyum on High Performance Architecture (HPCA), pp. 312-323, fevereiro, 1998.

[74] STAFFORD W. *The Berkeley Now Project.* Estados Unidos da América. Disponível em <http://now.cs.berkeley.edu>. Acesso em janeiro de 1998.

[75] STUM M; ZHOU S. *Algorithms Implementing Distributed Shared Memory*, artigo de revista, IEEE Computer v.23 , n.5 , pp.54-64, maio, 1990.

[76] SWANSON M.; STOLLER L.; CARTER J. *Making Distributed Shared Memory Simple, Yet Efficient.* Technical report. Laboratório de Sistemas de Computação, Universidade de Utah, 1998.

[77] TANEMBAUM A.S., *Distributed Operating Systems.* Livro. Primeira Edição, Prentice Hall, 1995.

[78] *Task Force on Cluster Computing* (IEEE-TFCC). Disponível em <http://www.ieeetfcc.org>. Acesso em janeiro de 2001.

[79] *The Beowulf Underground.* Estados Unidos da América. Disponível em <www.beowulf-underground.org/documentation.html>. Acesso em setembro de 1997.

[80] THITIKAMOL K.; KELEHER P. *Multi-Threading and Remote Latency in Software DSMs.* Artigo em evento. The 17th International Conference on Distributed Computing Systems, maio, 1997.

[81] UENG J. C.; SHIEH C. K.;LIN Q.C. *Design and Implementation of Proteus.* Artigo em evento. 1st Workshop on Software Distributed Shared Memory (WSDSM '99), Rhodes, Grécia, 25 de junho, 1999.

[82] *Vote-Communicatiion Support System.* Instituto de Arquitetura de Computadores e Tecnologia de Software. Berlin. Alemanha. Disponível em <http://www.first.gmd.de/vote >. Acesso em outubro de 1999.

[83] WEIWU H. *Reducing Message Overhead in Home Based Software DSMs.* Artigo em evento. Proceedings of ACM 1st Workshop on Software DSM System, Grécia, junho, 1999.

[84] WELSH M. ;EICKEN T. V.; Low-Latency Commnunication over Fast-Ethernet, Technical Report, Universidade de Berkeley, 1998.

[85] *Wisconsin Wind Tunnel Project.* Universidade de Wisconsin. Estados Unidos da América. Disponível em <http://www.cs.wisc.edu/~wwt>. Acesso em dezembro de 1997.

[86] WOO S.; OHARA M.; TORRIE E.; SINGH J. P.; GUPTA A. *The SPLASH-2 programs: Characterization and methodological considerations.* artigo de revista, *In Proceedings of the 22th Annual Symposium on Computer* Architecture, pp. 24-36, junho, 1995.

[87] ZHOU Y; IFTODE L.; LI K. *Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems.* Artigo de evento. Proceedings of the 2nd Symposium on Operating Systems Design and Implementation, outubro, 1996.

Capítulo 7

Apêndice A - Análise Detalhada

7.1 Análises Comparativas

Nesta seção será feita a análise comparativa detalhada de todos os programas (executados nos DSMs) avaliados neste trabalho.

7.1.1 EP (NAS)

Os *speedups* do aplicativo EP podem ser vistos na figura 7.1, onde são mostrados os *speedups* dos três DSMs para três diferentes parâmetros de entrada M^{24}, M^{26}, M^{28}.

Como se pode perceber pela figura 7.1, o TreadMarks é mais rápido que o JIAJIA e que o Nautilus, exceto para M^{28}, quando o JIAJIA é mais rápido que ambos. Como se pode perceber pela tabela 5.2, em valores numéricos, para 16 nós, a diferença de *speedup* entre esses DSMs varia de 8,3% a 16,6% para M^{24}, de 0,89% a 15,7% para M^{26} e de 0,1% a 15,5% para M^{28}.

Comparando-se o TreadMarks com o JIAJIA em termos de número de mensagens, o JIAJIA transmite 14,1% em média menos mensagens para M^{24}, M^{26} e M^{28}. Em respeito à quantidade de kbytes, como pode ser observado na tabela 7.2, o JIAJIA transmite em volta de 9 vezes mais que o TreadMarks. Na tabela 7.3 nota-se que o JIAJIA transmite o dobro do número de páginas. A maior quantidade de kilobytes e páginas transmitidos sobrecarrega mais a rede, diminuindo assim o desempenho do JIAJIA. Para M^{26} e M^{28}, os *speedups* do TreadMarks e do JIAJIA ficaram bem próximos, devido ao aumento de localidade pelo incremento do parâmetro de entrada. Com

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
msgs.Tmk.4096	297	297	297	24590	79339	200052	34845	121309	311494	10427	14923	27109	108976	1171427
msgs.JIA.4096	255	255	255	20764	68146	177562	29034	94142	276847	12168	-	28880	107017	85291
msgs.Naut.4096.n	365	320	320	23092	74730	201558	10872	42084	123947	4968	6400	5032	36200	81988

Tabela 7.1: Número de mensagens para 16 nós

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
kB.Tmk.4096	91	91	91	17023	46360	107326	69620	243011	624618	16428	24320	48039	31681	131614
kB.JIA.4096	970	970	970	10574	29210	64975	57056	191437	563284	22597	-	65030	137895	162715
kB.Naut.4096.n	56	56	56	29850	121507	374428	43834	168230	495658	8682	11748	8760	59375	167242

Tabela 7.2: Número de kbytes para 16 nós

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
page.Tmk.4096	15	15	15	1820	5460	18790	17325	60542	155599	3963	5912	11780	556	6586
page.JIA.4096	30	30	30	8342	30638	82946	14967	47056	138416	5396	-	13728	32711	44570
page.Naut.4096.n	15	15	15	8359	32072	91669	10840	42058	123915	2003	2497	1885	5383	39043

Tabela 7.3: Número de mensagens de páginas transmitidas para 16 nós

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
SIGS.Tmk.4096	15	15	15	266	785	2239	960	3360	8640	3813	5725	11478	11?	5834
SIGS.JIA.4096	2	2	2	8342	30638	82946	14967	47056	138416	33236	63264	105888	49011	87189
SIGS.Naut.4096.n	15	15	15	18604	87164	329325	1024	4480	9216	42943	76267	124687	9895	85569

Tabela 7.4: Número de SIGSEGVs para 16 nós

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
diffs.Tmk.4096	15	15	15	2639	9914	24655	0	0	0	868	1167	1362	15142	46220
diffs.JIA.4096	15	15	15	0	0	0	0	0	0	343	-	367	6375	348
diffs.Naut.4096.n	15	15	15	0	0	0	0	0	0	300	300	300	3368	1898

Tabela 7.5: Número de mensagens de *diffs* para 16 nós

respeito ao número de SIGSEGVs, que pode ser visto na tabela 7.4, o JIAJIA apresenta um número 7,5 vezes menor que o TreadMarks, portanto menos mudanças de estado de páginas que podem ser justificadas pela maior transferência (kB). Com respeito ao número de mensagens de *diffs*, ambos empatam conforme os resultados exibidos na tabela 7.5. Concluindo, pela maior quantidade de mensagens e kbytes transmitidos, o JIAJIA acaba sendo mais lento que o TreadMarks para este *benchmark*.

A diferença de *speedup* entre o TreadMarks e Nautilus pode ser justificada pela maior quantidade de mensagens (até 18,6% mais como mostrado na tabela 7.1), e pela implementação dos semáforos distribuídos do Nautilus, que precisa ser melhorada. Com respeito ao número de kbytes, o Nautilus transmite 38,5% menos, como pode ser notado na tabela 7.2. Em termos de número de mensagens de páginas, de número de SIGSEGVs e número de mensagens de *diffs*, ambos empatam, como pode ser observado nas tabelas 7.3, 7.4 e 7.5.

7.1.2 LU (SPLASH-2)

Como se pode perceber pela figura 5.1, o comportamento geral dos gráficos é o seguinte: até 8 nós o JIAJIA é mais rápido que os outros, enquanto que para mais de 8 nós, o TreadMarks é mais rápido que o JIAJIA e este, mais rápido que o Nautilus, exceto para tamanho de matriz igual a 1792. Com 16 nós, o desempenho do Nautilus fica praticamente o mesmo do JIAJIA. Como se pode perceber na tabela 5.2, em valores numéricos, para 16 nós, a diferença de *speedup* entre o TreadMarks e o JIAJIA e o Nautilus chega a respectivamente 5,3% e 6,3% para tamanho de matriz 1024, 3,2% e 3,0% para 1792 e, 2,1% e 2,5% para 3072. Deve-se observar também que à medida que se aumenta o tamanho das matrizes de entrada, mais altos ficam os *speedups* devido à melhora da localidade dos programas. Para 1792, a distribuição de dados desfavorece o Nautilus, o que acarreta uma perda de desempenho.

Uma observação geral importante é que o LU é um aplicativo do tipo *multiple reader single writer*, onde a segunda fase de computação é usada por todos os processadores para atualizar uma das submatrizes envolvidas no programa. Neste caso, o TreadMarks irá necessitar de vários *diffs* para atualizar uma página, enquanto que o JIAJIA e o Nautilus necessitarão apenas da página que está localizada no *home*.

Como se pode observar nas tabelas 7.1, 7.2, 7.3, 7.4 e 7.5 para 16 nós, o JIAJIA, quando comparado ao TreadMarks, transmite menos mensagens (15,6%, 14,1% e 11,2% para as entradas 1024, 1792 e 3072) e menos quantidade de kbytes pela rede (37,9%, 37,0% e 39,5% para as entradas 1024, 1792 e 3072). O JIAJIA não produz *diffs* porque seu protocolo de coerência sendo *home-based* não produz *diffs* para atualizações de páginas que estão nos nós *home*; porém o JIAJIA devido a seu modelo de consistência de escopo com nós *home,* transmite uma quantidade bem maior de páginas (4,58, 5,61 e 4,41 vezes mais para os tamanhos 1024, 1792 e 3072), do que o TreadMarks, que por sua vez transmite 2639, 9914 e 24655 *diffs* para os respectivos 1024, 1792 e 3072 tamanhos de matriz de entrada, característica esta do modelo de consistência de liberação

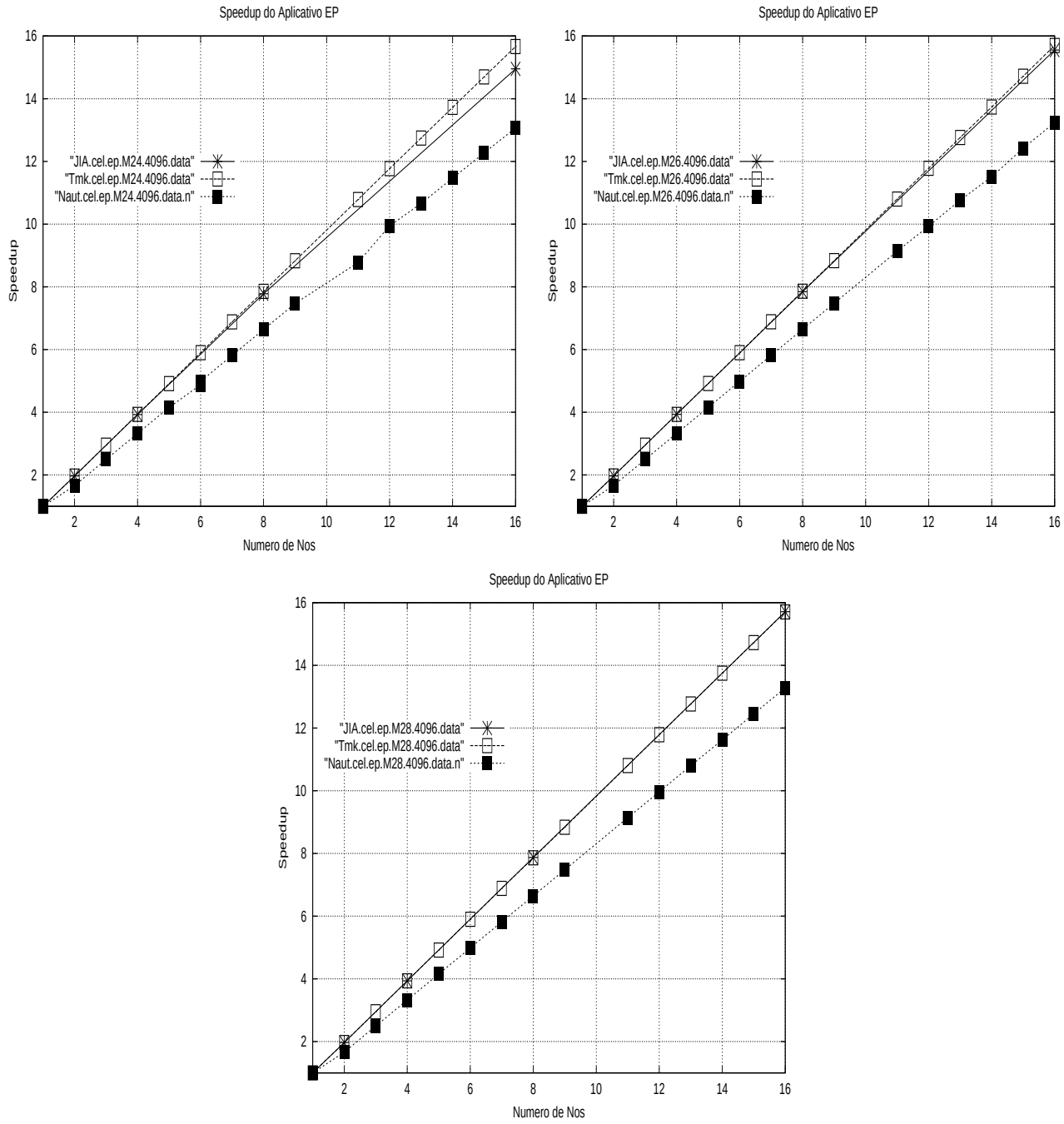

Figura 7.1: *Speedups* do aplicativo EP (NAS) - M^{24}, M^{26}e M^{28}

preguiçosa. O maior valor numérico de páginas transmitidos pelo JIAJIA contra as mensagens de *diffs* do TreadMarks sobrecarrega mais a rede, diminuindo o desempenho do JIAJIA e também ocupando mais banda nas saídas das barreiras. Portanto, mesmo com um menor número de kbytes e um menor número global de mensagens, o TreadMarks, devido à adoção do modelo de consistência de liberação preguiçosa que maximiza a utilização de *diffs*, minimizando o tráfego de páginas, acabando por superar o JIAJIA.

Comparando o TreadMarks e Nautilus em termos de número de mensagens, para os tamanhos de matriz de 1024 e 1792, o Nautilus transmite 6,1% e 5,8% menos mensagens, enquanto que para o tamanho de 3072, o TreadMarks transmite 0,7% menos. Com respeito à quantidade de kbytes transmitida, o Nautilus transmite 1,8, 2,6 e 3,5 vezes mais kbytes que o TreadMarks para os tamanhos de entrada 1024, 1792 e 3072. A transmissão de maior quantidade de kbytes pode ser justificada pela maior quantidade de páginas transmitidas (4,6, 2,6 e 4,9 vezes mais para os tamanhos 1024, 1792 e 3072); este parâmetro acabou sendo o principal responsável pelo menor *speedup* do Nautilus quando comparado ao TreadMarks. Graças à adoção do modelo de consistência de escopo com a característica de *home-based*, o Nautilus também não produz *diffs* para atualizações de páginas que estão no nó *home* e, desta forma, o Nautilus transmite menos *diffs* que o TreadMarks (nenhum *diff* do Nautilus contra 2639, 9914 e 24655 do TreadMarks para os respectivos 1024, 1792 e 3072 de matriz de entrada).

Com relação ao número de SIGSEGVs (ou SIGs), o JIAJIA e o Nautilus apresentam um número bem maior que o TreadMarks, isto é, 31,3%, 39,2% e 37,0% para o JIAJIA e 70,0%, 111,0% e 147,1% para o Nautilus, plenamente justificado pela utilização maior de páginas (pelo JIAJIA e Nautilus) ao invés de *diffs* (pelo TreadMarks).

Pela adoção do mesmo modelo de consistência (de escopo) usado pelo JIAJIA, o Nautilus deveria ter um comportamento semelhante com relação ao número de mensagens e quantidade de *kilobytes*. Isto não ocorre devido à diferente distribuição de dados, que causa uma maior quantidade de mensagens geradas pelo falso compartilhamento, prejudicando assim o desempenho do Nautilus.

7.1.3 MM

Como se pode observar nas tabelas 7.1, 7.2, 7.3, 7.4 e 7.5 para 16 nós, o Nautilus transfere 2,8, 2,9 e 2,5 vezes menos mensagens que o TreadMarks e 2,7, 2,2 e 2,2 vezes menos que o JIAJIA (para os tamanhos de 1024, 1792 e 3072). Com relação à quantidade de kbytes, o Nautilus transfere menos 37,0%, 30,8% e 20,6% menos que o Treadmarks e 23,2%, 12,1% e 12,0% menos que o JIAJIA para os tamanhos de 1024, 1792 e 3072. Em respeito ao número de mensagens de páginas, o Nautilus transmituiu 35,7%, 30,6% e 20,4% menos que o TreadMarks e 27,6%, 10,6% e 11,7% menos que o JIAJIA para os tamanhos de 1024, 1792 e 3072. A distribuição de dados adotada como padrão para todos os testes permitiu que no Nautilus todos os nós possuíssem mais páginas localmente ser ter a necessidade de trafegá-las; deste modo o tempo de *cold-startup* para este aplicativo é

menor no Nautilus. Não se deve esquecer ainda que a não utilização do sinal SIGIO para indicar a presença de uma mensagem que chega e o *multi-threading* minimizam os *overheads*, permitindo maiores *speedups*. Todos estes fatores diminuíram a sobrecarga da rede, melhorando o desempenho.

Confrontando o TreadMarks com o JIAJIA, o JIAJIA transmite menos mensagens em relação ao TreadMarks (16,7%, 22,4% e 12,5%), menos kilobytes (18,0%, 21,2% e 9,8%) e menos páginas (13,6%, 22,3% e 11,0% para tamanhos de entrada de 1024, 1792 e 3072, respectivamente). Todos estes parâmetros acabariam favorecendo o JIAJIA em termos de *speedup*, contudo os *speedups* do TreadMarks foram melhores, devido a um possível problema de implementação. Além disso, o número de SIGs é uma ordem de magnitude maior no JIAJIA, o que acaba prejudicando seu desempenho.

Convém também salientar que nenhum dos DSMs envolvidos nesta avaliação transmitiu mensagens de *diffs* para nenhum dos três parâmetros de matriz de entrada.

7.1.4 SOR (Rice-University)

Como pode ser percebido pela figura 5.3 e pela tabela 5.2, os *speedups* do JIAJIA (que não foram plotados) não são muito usuais, e portanto não vão ser considerados para comparação. Acredita-se haver algum problema de implementação deste DSM com o algoritmo do SOR.

Considerando a tabela 5.2, para 16 nós, pode-se considerar que o Nautilus é mais rápido que o TreadMarks (4,4% e 27,0% respectivamente para tamanhos de matriz de 1024 e 2048), exceto para tamanho de matriz 1536, quando o TreadMarks é 10,8% mais rápido que o Nautilus. Neste caso, a distribuição de dados causa um maior efeito do falso compartilhamento, o que prejudica o desempenho do Nautilus. De maneira geral somente em alguns pontos, como se pode perceber especificamente para número de nós igual a 11 da figura 5.3 a e para número de nós menor que 5 da figura 5.3c, o TreadMarks supera o Nautilus.

Pode-se justificar o comportamento geral mencionado observando as tabelas 7.1, 7.2, 7.3, 7.4 e 7.5, pois o número de mensagens transmitidas pelo Nautilus é menor que o TreadMarks (52,4%, 57,1% e 81,4% para 1024, 1536 e 2048), o número de kilobytes transmitidos é menor (47,2%, 57,8% e 84,0% para 1024, 1536 e 2048), o número de páginas transmitidas é menor (49,6%, 57,8% e 84% para para 1024, 1536 e 2048), e também porque o número de *diffs* transmitidos é menor (65,4%, 74,3% e 78,0% para 1024, 1536 e 2048). Pode-se justificar o número menor de mensagens, de kilobytes e de *diffs* porque o *overhead* para manter o diretório numa falta de página, a forma de produzir as *write-notices*, o modelo de consistência de escopo *home-based*, todos contribuem para o melhor desempenho do Nautilus. Além disso a distribuição de dados faz com que o *cold-startup* do Nautilus seja menor, bem como o *multithreading* e a não utilização do sinal SIGIO (indica a chegada de mensagem) diminuem os *overheads* melhorando seu desempenho.

7.1.5 Water (SPLASH-2)

Na figura 7.2, podem ser vistos os *speedups* do aplicativo Water para os vários DSMs com um único conjunto de parâmetros de entrada: 1728 moléculas e 25 passos.

Como se pode perceber, os *speedups* do TreadMarks e do JIAJIA são próximos até 16 nós. O TreadMarks supera o JIAJIA em 4,9%. Como pode ser observado na tabela 7.1, o TreadMarks transmite 1,8% mais mensagens que o JIAJIA. Em termos de número de kilobytes transmitidos, o JIAJIA transmite 4,4 vezes maior que o do TreadMarks (tabela 7.2), prejudicando assim seu desempenho. Como pode ser averiguado na tabela 7.3, o número de páginas transmitidas pelo JIAJIA é 58,8 vezes maior que o TreadMarks. Com respeito ao número de SIGSEGVs, o JIAJIA apresenta duas ordens de magnitude mais do que o TreadMarks. Em contrapartida, o JIAJIA transmite 57,9% menos mensagens de *diffs* que o TreadMarks (tabela 7.5), o que pode ser justificado pelo modelo de consistência de liberação preguiçosa do TreadMarks que transmite *diffs* para atualizar páginas. A pequena diferença de *speedup* entre o TreadMarks e JIAJIA se deve ao fato deste transmitir um número maior de páginas, conseqüentemente de kbytes, o que acaba sobrecarregando a rede, diminuindo seu *speedup*.

Comparando-se o TreadMarks com o Nautilus, o TreadMarks o supera em 0,5%, como pode ser notado por meio da figura 7.2 e da tabela 5.2. Em termos de número de mensagens, a tabela 7.1 mostra que o Nautilus transmite 3,0 vezes menos mensagens, devido a distribuição de dados mais adequada e ao menor *cold-startup* por este apresentado. Com respeito ao número de kbytes, que pode ser visto na tabela 7.2, o Nautilus transmite 46,6% mais kbytes, o que acaba sendo o fator responsável pelo maior *speedup* do TreadMarks. Estes 46,6% mais kbytes são relativos a páginas, já que na tabela 7.3, pode-se notar que o Nautilus transmite 9,7 vezes mais páginas que o TreadMarks, devido às diferenças existentes entre os modelos de consistência adotados por estes DSMs (de liberação preguiçosa e de escopo). Em se tratando da quantidade de SIGSEGVs, o Nautilus apresenta duas ordens de magnitude mais sinais devido à necessidade de se atualizar as páginas. E por fim, conforme era esperado, o TreadMarks transmite 4,5 vezes mais mensagens de *diffs* que o Nautilus, diferença esta devido aos diferentes modelos de consistência por eles adotados. Concluindo, apesar do Nautilus transmitir menos mensagens, a quantidade de kbytes devida a páginas, acaba sendo responsável pela pequena diferença de *speedup* entre ambos, ou seja, no fundo a diferença ocorre devido ao modelo de consistência adotado por cada um deles.

7.1.6 Barnes (SPLASH-2)

Na figura 5.4 podem ser vistos os *speedups* dos três DSMs sob o aplicativo Barnes para um único conjunto de parâmetros de entrada: 16384 corpos gravitacionais.

Pode-se dizer que o Nautilus foi mais rápido que o TreadMarks à medida que o número de nós foi incrementado. Para 16 nós, pela tabela 5.2, pode-se dizer que o Nautilus foi 3,54% mais rápido que o TreadMarks e 62,53% mais rápido que o JIAJIA.

Observando-se as tabelas 7.1, 7.2, 7.3, 7.4 e 7.5, o TreadMarks transmitiu 14,3 vezes mais

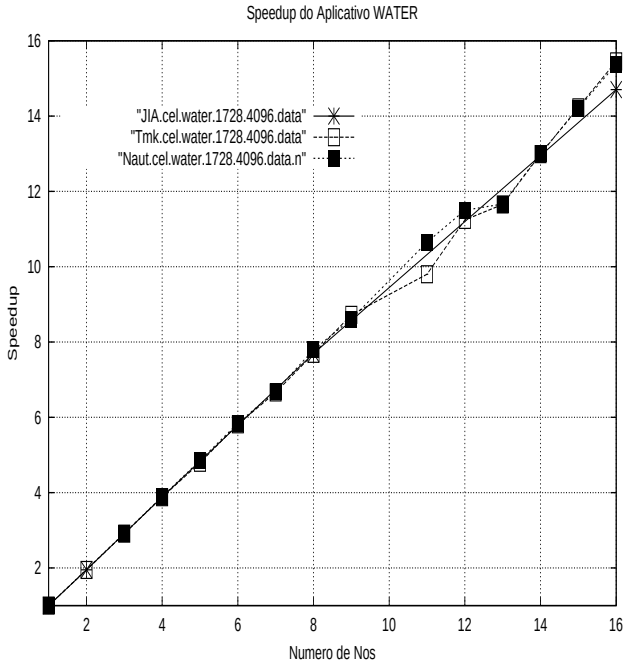

Figura 7.2: *Speedups* do aplicativo Water (SPLASH-2) - 1728 moléculas

mensagens que o Nautilus e 13,7 vezes mais que o JIAJIA. O TreadMarks transmite 19,1% menos kilobytes que o JIAJIA e 29,6% menos kilobytes que o Nautilus. O JIAJIA transmite 6,8 vezes mais páginas que o TreadMarks e o Nautilus transmite 5,9 vezes mais páginas que o TreadMarks. Em compensação o TreadMarks, pela adoção do modelo *lazy release*, acaba sofrendo para atualizar as páginas quando ocorrem as faltas de página, isto é, para a coleta de *diffs*: transmite 132,8 vezes mais mensagens que o JIAJIA (duas ordens de magnitude) e 24,4 vezes mais *diffs* que o Nautilus (uma ordem de magnitude). O maior número de *diffs* e, conseqüentemente, a maior complexidade para armazená-los e manipulá-los, acaba sendo o grande responsável pelo pior desempenho do TreadMarks em relação ao Nautilus.

A diferente implementação, estando aí inclusos o *multithreading* e a não utilização do SIGIO, bem como uma distribuição de dados que minimiza o tempo de *cold-startup*, também favoreceram o Nautilus quando comparado ao JIAJIA.

7.2 Análises Comparativas: Técnica de Agregação de Páginas

Nesta seção será feita a análise detalhada da técnica de agregação de páginas.

7.2.1 EP (NAS)

Nas figuras 7.3, 7.4 e 7.5 os *speedups* do aplicativo EP podem ser vistos para os vários DSMs, com o Nautilus utilizando vários tamanhos de página.

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
msgs.Tmk.4096	297	297	297	24590	79339	200052	34845	121309	311494	10427	14923	27109	108976	1171427
msgs.JIA.4096	255	255	255	20764	68146	177562	29034	94142	276847	12168	-	28880	107017	85291
msgs.JIA.8192	255	255	255	14098	46538	153166	13980	46531	138348	7640	9582	19902	85069	45393
msgs.JIA.16384	255	255	255	10786	30410	104674	7020	22636	69003	7640	9582	19902	72156	23967
msgs.JIA.32768	255	255	255	9130	22286	62986	3540	11453	33901	7640	9582	19902	65874	13171
msgs.Naut.4096.n	365	320	320	23092	74730	201558	10872	42084	123947	4968	6400	5032	36200	81988
msgs.Naut.8192.n	332	320	320	16686	50355	165299	5354	20629	61732	3456	4384	5318	43200	44217
msgs.Naut.16384.n	334	318	327	13517	34327	113292	2582	10244	30783	2490	2764	3466	52800	24685
msgs.Naut.32768.n	367	318	320	13477	27238	71717	1271	5062	15382	2560	4200	2456	49600	16317

Tabela 7.6: Número de mensagens para 16 nós - Técnica de Agregação de Páginas

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
kB.Tmk.4096	91	91	91	17023	46360	107326	69620	243011	624618	16428	24320	48039	31681	131614
kB.JIA.4096	992	992	992	10574	29210	64975	57056	191437	563284	22597	-	65030	137895	162715
kB.JIA.8192	255	255	255	10774	31686	82404	54255	187111	557929	21664	31671	75761	179760	169495
kB..JIA.16384	501	501	501	11779	31446	94414	54008	181217	553867	38546	56234	73922	249666	169255
kB.JIA.32768	992	992	992	14033	33384	94605	53855	182302	542416	72312	105360	137106	393310	168914
kB.Naut.4096.n	56	56	56	29850	121507	374428	43834	168230	495658	8682	11748	8760	59375	167242
kB.Naut.8192.n	107	107	107	39102	140491	593556	42581	164772	493599	9375	13437	20312	62500	172473
kB.Naut.16384.n	227	227	227	55687	198801	764739	40811	163398	492012	9843	15000	21093	140625	166892
kB.Naut.32768.n	452	452	452	96838	235464	813899	39647	162876	491200	10234	19062	19687	234000	209164

Tabela 7.7: Número de kbytes transmitida para 16 nós - Técnica de Agregação de Páginas

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
page.Tmk.4096	15	15	15	1820	5460	18790	17325	60542	155599	3963	5912	11780	556	6586
page.JIA.4096	30	30	30	8342	30638	82946	14967	47056	138416	5396	-	13728	32711	44570
page.JIA.8192	30	30	30	5054	19834	70748	7200	23288	69159	3132	4092	9236	21737	23184
page.JIA.16384	30	30	30	3398	11770	46502	3600	11300	34479	3132	4092	5052	15280	11753
page.JIA.32768	30	30	30	2570	7708	25658	1800	5641	16928	3132	4092?	5052	12139	5996
page.Naut.4096.n	15	15	15	8359	32072	91669	10840	42058	123915	2003	2497	1885	5383	39043
page.Naut.8192.n	15	15	15	5114	19849	73655	5322	20597	61700	1077	1515	2003	5680	19151
page.Naut.16384.n	15	15	15	3458	11714	47582	2550	10212	30751	570	837	1077	7290	10434
page.Naut.32768.n	15	15	15	3123	8001	26818	1239	5090	15350	1185	1453	572	6115	6250

Tabela 7.8: Número de mensagens de páginas transmitidas para 16 nós - Técnica de Agregação de Páginas

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
SIGS.Tmk.4096	15	15	15	266	785	2239	960	3360	8640	3813	5725	11478	11?	5834
SIGS.JIA.4096	45	45	45	8342	30638	82946	14967	47056	138416	33236	63264	105888	49011	87189
SIGS.JIA.8192	45	45	45	5054	19834	70748	7200	23288	69159	22332	32892	64916	33112	46133
SIGS.JIA.16384	45	45	45	3398	11770	46502	3600	11300	34479	22332	32892	43452	24255	23362
SIGS.JIA.32768	45	45	45	2570	25658	24152	1800	5641	16928	22332	32892	30382	19889	11851
SIGS.Naut.4096.n	15	15	15	18604	87164	329325	1024	4480	9216	42943	76267	124687	9895	85569
SIGS.Naut.8192.n	15	15	15	11796	49169	209133	1024	3136	6144	21737	47715	83903	10459	45804
SIGS.Naut.16384.n	15	15	15	7264	28117	119443	768	2352	4608	11060	24077	42217	12059	23009
SIGS.Naut.32768.n	15	15	15	5736	16440	63152	406	1185	3472	6011	12557	21300	11962	11699

Tabela 7.9: Número de SIGSEGVs para 16 nós - Técnica de Agregação de Páginas

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
diffs.Tmk.4096	15	15	15	2639	9914	24655	0	0	0	868	1167	1362	15142	46220
diffs.JIA.4096	15	15	15	0	0	0	0	0	0	343	354	367	6375	348
diffs.JIA.8192	15	15	15	0	0	0	0	0	0	343	354	370	6375	345
diffs.JIA.16384	15	15	15	0	0	0	0	0	0	343	354	370	6376	343
diffs.JIA.32768	15	15	15	0	0	0	0	0	0	343	354	370	6376	342
diffs.Naut.4096.n	15	15	15	0	0	0	0	0	0	300	300	300	3368	1898
diffs.Naut.8192.n	15	15	15	0	0	0	0	0	0	300	300	300	3691	1775
diffs.Naut.16384.n	15	15	15	0	0	0	0	0	0	300	300	300	4637	1892
diffs.Naut.32768.n	15	15	15	0	0	0	0	0	0	300	300	300	4602	1895

Tabela 7.10: Número de mensagens de *diffs* para 16 nós - Técnica de Agregação de Páginas

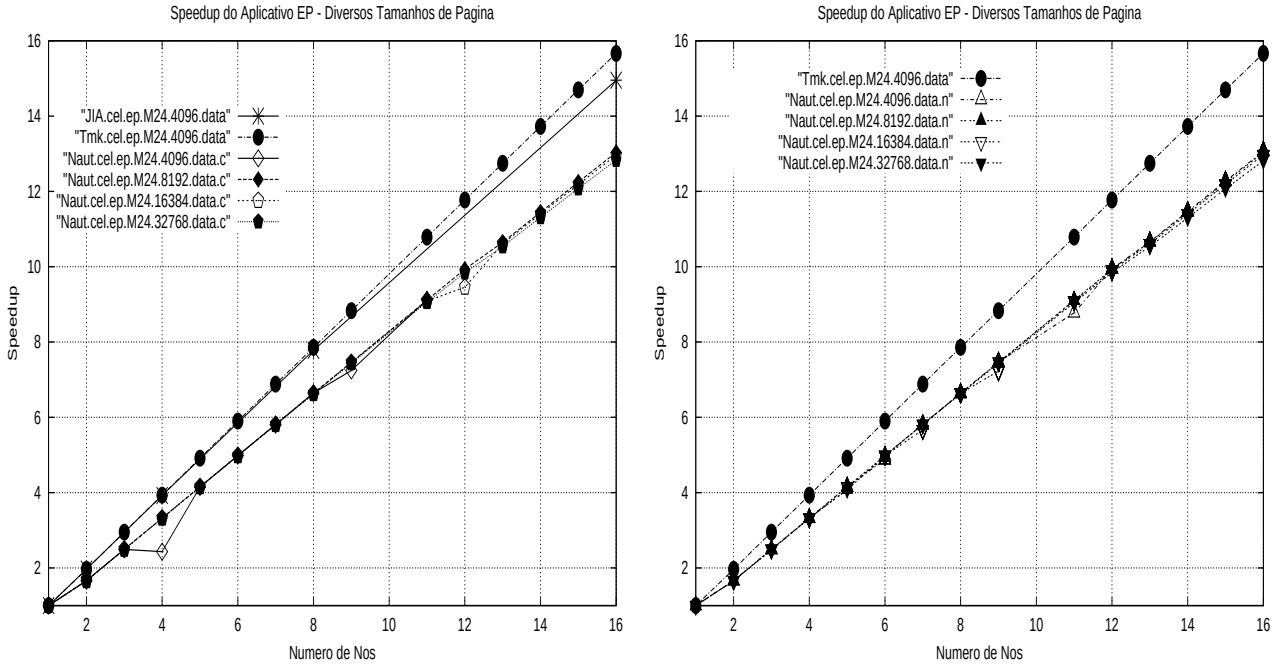

Figura 7.3: *Speedups* do aplicativo EP (NAS) - tamanho M^{24} -Técnica de Agregação de Páginas

Para o JIAJIA, pode-se perceber através dessa figura e da tabela 5.3, que os *speedups* praticamente não diminuem à medida que se aumenta o tamanho da página. Também na tabela 7.6, pode-se perceber que o número de mensagens se mantém constante. Pela tabela 7.7, percebe-se que o número de kbytes transmitidos aumenta graças ao aumento do efeito do falso compartilhamento conseqüente da aplicação da técnica de agregação. Observando a tabela 7.8, o número de páginas transmitidas permanece constante. Na tabela 7.9 percebe-se a não diminuição do número de SIGSEGV, que seria o comportamento resultante da aplicação da técnica de agregação de páginas. Pode-se justificar que a técnica de agregação não apresentou efeito prático nos *speedups* pois este *benchmark* praticamente não transfere dados. Por fim, também nota-se pela observação da tabela 7.10, o número de mensagens de *diffs* permanece o mesmo.

Comparando-se com a versão tradicional, percebe-se que com o aumento do tamanho de página, os *speedups* do Nautilus chegaram a diminuir de até 1,9%, como pode ser observado nas figuras 7.3, 7.4 e 7.5, e na tabela 5.3. Em respeito ao número de mensagens, como pode ser notado na tabela 7.6, este chegou a diminuir em alguns casos de até 8,5%. Com respeito à quantidade de kbytes, esta aumentou de até 8,1 vezes (tabela 7.7), como conseqüência direta aumento do tamanho da página, já que o número de mensagens de páginas, como pode ser visto na tabela 7.8, este se manteve constante. Pela tabela 7.9, a quantidade de SIGSEGVs se manteve, comportamento que já era esperado pela característica do aplicativo, e também já que o número de páginas se manteve. O número de mensagens de *diffs* também ficou o mesmo (tabela 7.10). Graças a pouca transferência de mensagens e de kbytes, que é uma característica do aplicativo EP, a técnica de agregação acaba não tendo influência sobre o *speedup* do Nautilus.

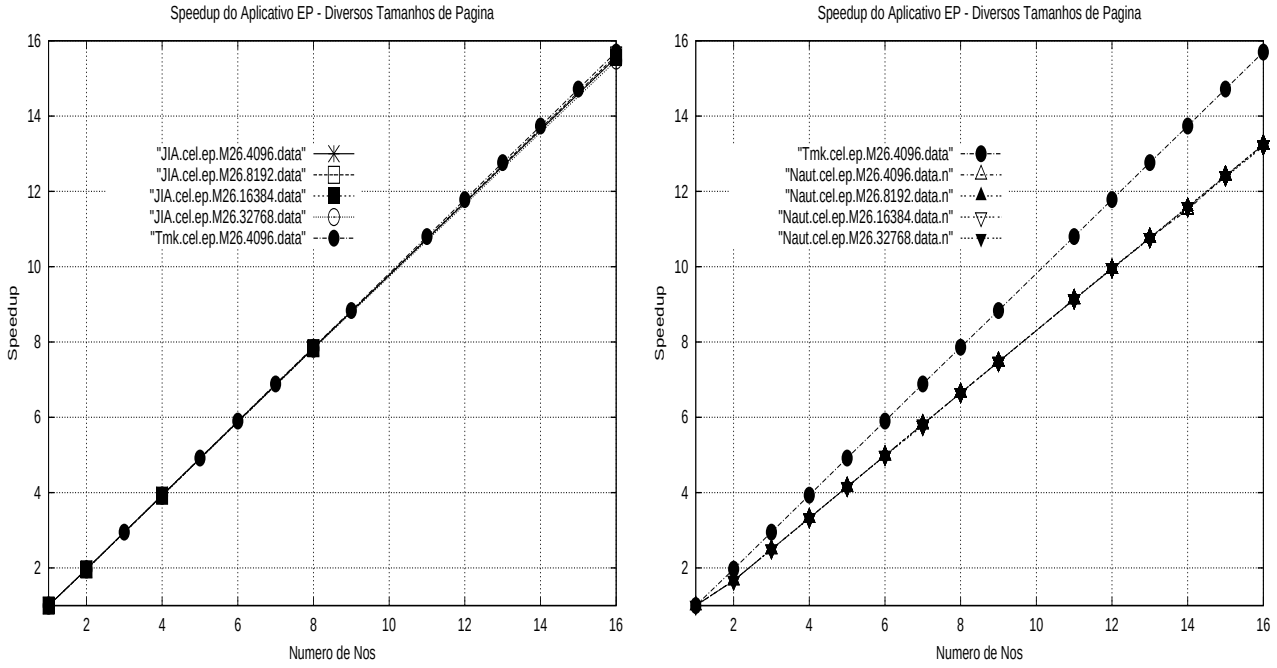

Figura 7.4: *Speedups* do aplicativo EP (NAS) - tamanho M^{26} - Técnica de Agregação de Páginas

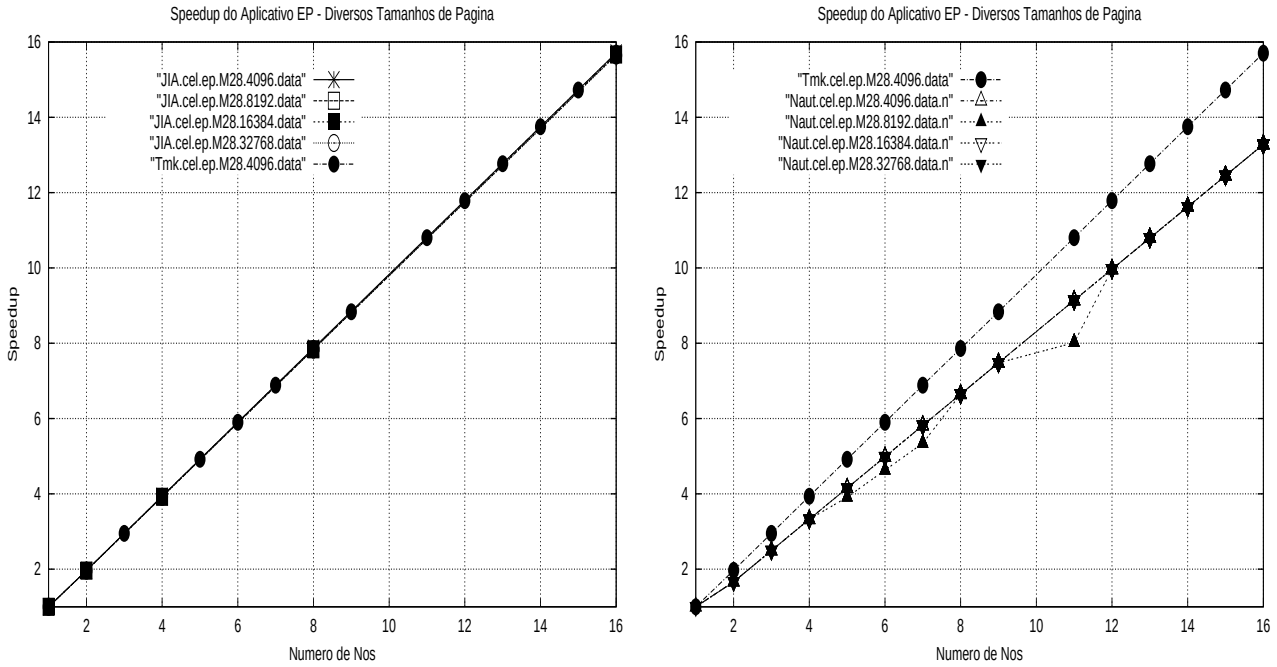

Figura 7.5: *Speedups* do aplicativo EP (NAS) - tamanho M^{28} - Técnica de Agregação de Páginas

7.2.2 LU (SPLASH-2)

Nas figuras 5.5, 5.6 e 5.7 estão apresentados os gráficos de *speedups* para o LU aplicando-se a técnica de agregação de páginas ao JIAJIA e Nautilus. Como se pode perceber, neste *benchmark* a técnica de agregação de página alterou significativamente, os *speedups* em alguns casos.

Para o JIAJIA, pode-se observar na tabela 5.3 e pelas figuras 5.5, 5.6 e 5.7 que houve uma pequena alteração de *speedup* (de no máximo 1,2%) ao se aplicar a técnica de agregação de páginas. Para tamanhos de página de 8kB e 16kB, o *speedup* do JIAJIA aumentou, caindo novamente ao mesmo nível de *speedup* de 4kB quando se utilizou página de 32kB. Pôde-se perceber também que à medida que o tamanho de página foi incrementado, de 4kB até 32kB, o número de mensagens caiu em até 67,3%, que é uma excelente redução do número de mensagens na rede. Em contrapartida, para os mesmos incrementos do tamanho de página, o número de kbytes aumentou de até cerca de 45,6%. O número de mensagens relativas a paginas foi reduzido de 69,2%, 74,6% e 69,1%, isto é, da mesma forma que se aumentou o tamanho da página de 4 vezes, o número de mensagens relativas a paginas também caiu de 4 vezes. Tendo havido uma boa redução do número de mensagens, pode-se concluir também o tamanho das mensagens aumentou, devendo-se levar em conta o aumento do tempo gasto para empacotá-las e desempacotá-las, que acabou refletindo no *speedup*. Outro fator que diminuiu sensivelmente foi o número de SIGs que caiu em torno de 69,1% a 70,9%, que era um efeito esperado da aplicação da técnica de agregação; no caso do tamanho 1792, devido ao aumento do falso compartilhamento, houve um aumento (118,0%) do número de SIGs quando se passou de 16kB para 32kB.

Para o Nautilus, pode-se observar pelas figuras 5.5, 5.6 e 5.7 e pelas tabelas 7.6, 7.7, 7.8 e 7.10, que à medida que se incrementa o tamanho de página, ocorre uma diminuição do *speedup* de até 14,4%, embora tenham havido saltos espúrios (crescentes) de *speedup*, por exemplo quando se estava usando páginas de 8kB com matriz de 1792 e páginas de 8kB com matriz de 3072. Com relação ao número de mensagens, à medida que se aumentou o tamanho de página de 4kB até 32kB, houve uma redução considerável de até 64,4% (tamanho de matriz de 3072 e páginas de 32kB). Em contrapartida, o número de kbytes que passam pela rede também foi aumentado em até 3,2 vezes mais. Com relação ao número de páginas que trafegam na rede, este foi reduzido de até 75,1%. A conseqűência imediata é o aumento do tamanho de mensagem, portanto tempo maior para empacotar, ler e desempacotar, conseqűentemente um gargalo para atender com rapidez o pedido por uma delas, fazendo com que se houvesse uma diminuição de *speedup* de até 14,4%. Uma conseqűência resultante da aplicação da técnica de agregação foi a diminuição (de 69,2% a 81,1%) do número de SIGs que pode ser observada na tabela 7.9.

7.2.3 MM

Nas figuras 5.8, 5.9 e 5.10 estão apresentadas as curvas de *speedup* do aplicativo MM para os DSMs quando se aplicou a técnica de agregação de páginas ao JIAJIA e ao Nautilus.

Pode-se observar por essa figura e pela tabela 5.3 que os *speedups* do JIAJIA aumentam à

medida que o tamanho de página é incrementado. Para tamanho de matriz de 1024, o ganho de *speedup* é de 58,6%; para tamanho de matriz de 1792 o ganho de *speedup* é da ordem de 20,0%; e finalmente para tamanho de matriz de 3072, o ganho de *speedup* é cerca de 13,9%. Na tabela 7.6, à medida que se aumenta o tamanho da página, a redução do número de mensagens ficou em torno de 87,8% para os vários tamanhos de matriz de entrada (1024, 1792 e 3072). À medida que se aumenta o tamanho da página, pode-se afirmar que o número de kbytes diminui como se pode observar na tabela 7.7 pois, obtiveram-se reduções de 5,6%, 4,8% e 3,7% respectivamente para os tamanhos 1024, 1792 e 3072. Com relação ao número de páginas, à medida que se aumenta o tamanho da página, o número de páginas foi reduzido de 88,0% (8 vezes que fica próximo ao aumento de 8 vezes o tamanho da página) como se pode observar na tabela 7.8. Estas reduções já eram esperadas como resultado da técnica de agregação. Devido à distribuição de dados adotada, o número de *diffs* para o JIAJIA foi nulo. Concluindo, com a redução do número de mensagens, do número de kbytes transferidos, do número de páginas, diminui-se o *overhead* sobre a rede e, conseqűentemente aumenta-se o desempenho. Pode-se perceber isto para por exemplo 16 nós e para tamanho de matriz 1024 e 1792, quando o TreadMarks supera em termos de *speedup* o JIAJIA para tamanho de página 4kB; mas para páginas de 32kB no primeiro caso (1024) e para páginas de 8kB, 16kB e 32kB no último caso (3072), o JIAJIA supera o TreadMarks. O número de SIGs também diminui em cerca de 88,0% (7.9), conseqűência direta da aplicação da técnica.

Por meio das figuras 5.8, 5.9 e 5.10 e da tabela 5.3, nota-se que os *speedups* do Nautilus praticamente não mudam (somente para tamanho de matriz de 1024, que os *speedups* melhoram em 2,0%). O número de mensagens é reduzido de 88,0% em média para os vários tamanhos de matriz de entrada. O número de kbytes, em contrapartida, foi reduzido de somente 9,6%, 3,2% e 0,9% para os tamanhos de 1024, 1792 e 3072. O número de mensagens de páginas também foi reduzido de 88,0% em média. O número de SIGSEGVs foi reduzido de 60,4%, 79,6% e 62,3%, resultado que era esperado da aplicação da agregação. Pode-se dizer que a técnica de agregação não melhorou o *speedup* do Nautilus porque, apesar da grande redução do número de mensagens como conseqűência direta da redução do número de páginas decorrente da aplicação da agregação, passou a ocorrer uma sobrecarga por pedidos relativos as mesmas páginas, além do tempo maior necessário ao seu empacotamento e desempacotamento.

O número de mensagens de consistência não sofreu alteração pois o aumento do tamanho das páginas não mudou as áreas de memória compartilhada que é a matriz resultado da multiplicação.

7.2.4 SOR (Rice-University)

Observando os resultados referentes ao Nautilus nas figuras mencionadas anteriormente e na tabela 5.3, para os tamanhos das matrizes de entrada de 1024 e 1536, à medida que se aumenta o tamanho da página de 4kB até 16kB, os *speedups* aumentam. Porém, para páginas de 32kB, o mesmo não ocorre. Analisando a tabela 7.6, o número de mensagens foi reduzido de 52,0% em média para os tamanhos de 1024, 1536 e 2048. A quantidade de kbytes aumenta de 17,9%, 62,3% e 124,7%, comparada à versão com 4kB, como conseqűência do efeito do falso compartilhamento. O número

de páginas, como pode ser observado na tabela 7.8, conforme resultado esperado pela aplicação da agregação, foi reduzido de 68,5% em média. Também o número de SIGs diminui de 84,5% em média, conseqüência direta da aplicação da técnica. Concluindo-se, pode-se afirmar que a redução da quantidade de mensagens relativas a páginas foi bastante significativa, e mesmo tendo um aumento da quantidade de kbytes transferida, acaba-se resultando um melhor *speedup*. Crê-se que para páginas de 32kB, o que ocorre é que o tamanho das mensagens aumenta bastante, e o tempo de encapsulamento e desencapsulamento acaba sendo consideravelmente grande e também o pedido pelas mesmas páginas acaba causando um congestionamento de pedidos pelas mesmas.

O número de mensagens de *diffs* manteve-se mais ou menos o mesmo já que as fronteiras geográficas das matrizes que são as áreas compartilhadas e que portanto devem ser atualizadas se mantêm (espacialmente) mesmo com o aumento do tamanho das páginas.

7.2.5 Water (SPLASH-2)

Na figura 7.6 pode-se ver os *speedups* do aplicativo Water para os vários DSMs, quando aplicada a técnica de agregação de páginas ao Nautilus.

Para o JIAJIA, observando-se a figura 7.6 e a tabela 5.3, à medida que se aumenta o tamanho da página, a partir de 8kB, os *speedups* do JIAJIA pioraram ligeiramente (em torno de 3,7%). Com relação ao número de mensagens, houve uma redução de até 38,4% quando se incrementou o tamanho da página, como pode ser constatado na tabela 7.6. Com relação ao número de kbytes (tabela 7.7), estes chegaram a cair a um quarto do valor tradicional (4kB) ao se usar páginas de 8kB. Com relação ao número de mensagens de páginas, houve uma redução de 62,9%, mostrada na tabela 7.8. O número de SIGSEGVs foi reduzido de até 59,4% (tabela 7.9). O número de mensagens de *diffs* se manteve (tabela 7.10). A primeira conclusão é que o tamanho das mensagens de páginas e de *diffs* aumentou, o que acarreta o aumento do tempo para processá-las e aplicá-las, e também pelo aumento do tempo de espera por uma página. Além disso, como o Water é um aplicativo cuja característica principal é o alto número de mensagens de sincronização, e não de mensagens contendo páginas, a diminuição do número de mensagens relativas a páginas obtida com a aplicação da técnica de agregação não foi suficiente para influir no desempenho.

Em relação ao Nautilus, como pode se visto na figura 7.6 e na tabela 5.3, à medida que se aumenta o tamanho da página os *speedups* caem 9,7% ao se passar de 4kB para 8kB, mas aumentam ligeiramente (cerca de 7,20%) quando se passa de 8kB para 16kB ou para 32kB. O número de mensagens, como pode ser visto na tabela 7.6, sofre um aumento de até 45,9%. Como pode ser notado na tabela 7.7, o número de kbytes aumentou de até 3,9 vezes. O número de páginas trafegadas aumentou de até 35,4%, justificado pelo aumento do falso compartilhamento. O número de SIGSEGVs, mostrado na tabela 7.9, sofre um aumento de até 21,9%, como conseqüência direta do aumento do número de páginas trafegadas. O número de mensagens de *diffs* foi aumentado de até 37,7% (tabela 7.10). Portanto a redução de *speedup* do Nautilus com a aplicação da agregação se deve ao aumento do efeito do falso compartilhamento, pelo aumento do número de páginas

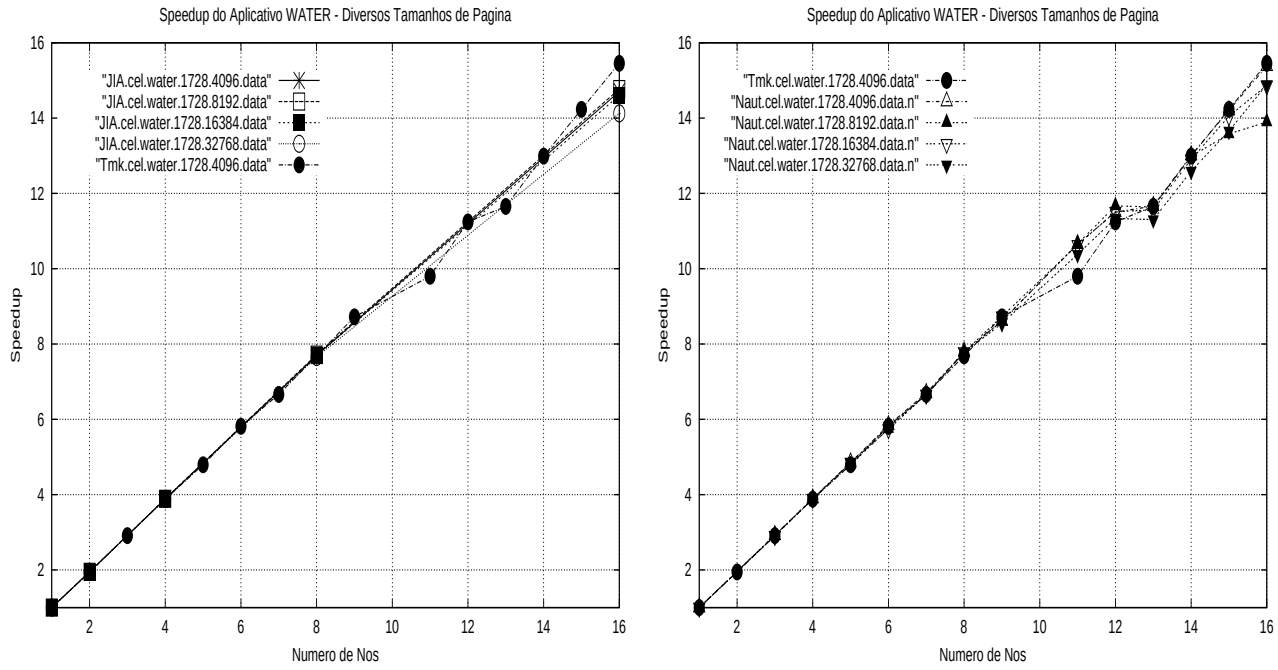

Figura 7.6: *Speedups* do aplicativo Water (SPLASH-2) - 1728 moléculas - Técnica de Agregação de Páginas

trafegadas (também de kbytes).

7.2.6 Barnes

Na figura 5.14 podem-se ver os *speedups* do aplicativo Barnes, quando submetido aos vários DSMs avaliados, com o JIAJIA e o Nautilus empregando a técnica de agregação de páginas.

Pode se observar na figura 5.14 e na tabela 5.3 que, para o JIAJIA, os *speedups* aumentaram 14,4% à medida que se aumenta o tamanho da página (4kB a 32kB). Com relação ao número de mensagens, como pode ser visto na tabela 7.6 houve uma redução de 84,6% quando se passou de 4kB para 32kB, que é uma conseqüência direta da redução do número de páginas trafegadas (tabela 7.8), em torno de 87,4%. Outra conseqüência direta foi uma diminuição sensível do número de SIGs de até 86,3%, como se pode observar na tabela 7.9. Em se tratando do número de kbytes, que pode ser observado na tabela 7.7, houve um incremento de 3,8% ao se aplicar a agregação de páginas. O número de mensagens de consistência (tabela 7.10) manteve-se mais ou menos o mesmo, para todos os tamanhos de página. A diminuição do número global de mensagens devido primordialmente à redução da parcela número de mensagens de páginas, conseqüências da agregação, justificam o aumento de *speedup* do JIAJIA para este aplicativo.

Para o Nautilus, como se pode constatar na figura 5.14 e na tabela 5.3, os *speedups* aumentaram quando se passou de 4kB para 8kB em torno de 3,9%, mantendo-se quando se passou de 8kB para 16kB, e diminuindo (de 7,2%) quando se passou de 16kB para 32kB. O número de mensagens, que pode ser visto na tabela 7.6, caiu de 80,0%. Observando-se o número de kbytes na tabela 7.7, este aumentou de até 4,2%, como aumento do efeito do falso compartilhamento. O número de páginas

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
msg.Tmk.4096	297	297	297	24590	79339	200052	34845	121309	311494	10427	14923	27109	108976	1171427
msg.JIA.4096	255	255	255	20764	68146	177562	29034	94142	276847	12168	-	28880	107017	85291
msg.Naut.4096.n	365	320	320	23092	74730	201558	10872	42084	123947	4968	6400	5032	36200	81988
msg.Naut.4096.c	-	319	320	-	21393	40898	10752	41971	123657	4508	5142	3114	54819	81645

Tabela 7.11: Número de mensagens para 16 nós - Técnica de Detecção de Escrita CO-WD

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
kB.Tmk.4096	91	91	91	17023	46360	107326	69620	243011	624618	16428	24320	48039	31681	131614
kB.JIA.4096	970	970	970	10574	29210	64975	57056	191437	563284	22597	-	65030	137895	162715
kB.Naut.4096.n	56	56	56	29850	121507	374428	43834	168230	495658	8682	11748	8760	59375	167242
kB.Naut.4096.c	-	56	56	-	19879	64431	42880	167756	494500	6556	8082	3881	34839	170948

Tabela 7.12: Número de kbytes transmitida para 16 nós - Técnica de Detecção de Escrita CO-WD

que trafegam na rede diminuiu de 83,9% quando se aplicou a técnica de agregação de páginas, como pode ser constatado na tabela 7.8, sendo responsável pela redução do número de mensagens. O número de SIGs diminuiu de 86,60%, resultado que também era esperado da aplicação da técnica de agregação. O número de mensagens de consistência manteve-se o mesmo ao se variar o tamanho da página, que pode ser observado na tabela 7.10, pois as áreas de compartilhamento não mudam geograficamente com a mudança do tamanho de página. Concluindo-se, com a aplicação da técnica de agregação, o número de mensagens foi reduzido, porém o número de kbytes aumentou, ou seja, o tamanho das mensagens aumentou. No caso de tamanho de página de 8kB, a redução do número de mensagens compensou e conseguiu-se um aumento de *speedup*; em contrapartida, com tamanho de mensagens de 16kB e principalmente de 32kB, houve o aumento do tempo de tratamento de mensagens e do número de kB, conseqüentemente quedas de *speedup*.

7.3 Análise Comparativa: Técnica de Detecção de Escrita CO-WD

Nesta seção será feita a análise comparativa da técnica de detecção de escrita CO-WD.

7.3.1 EP (NAS)

Na figura 7.7 os *speedups* do aplicativo EP podem ser vistos para os vários DSMs, com o Nautilus usando a técnica de detecção de escrita CO-WD.

Pode-se perceber nessa figura, assim como na tabela5.4 que os *speedups* do Nautilus não se modificam com a aplicação da detecção CO-WD. Como pode ser visto na tabela 7.11, o número de mensagens se manteve. Através da tabela 7.12, pode-se perceber que o a quantidade de kbytes

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
page.Tmk.4096	15	15	15	1820	5460	18790	17325	60542	155599	3963	5912	11780	556	6586
page.JIA.4096	30	30	30	8342	30638	82946	14967	47056	138416	5396	-	13728	32711	44570
page.Naut.4096.n	15	15	15	8359	32072	91669	10840	42058	123915	2003	2497	1885	5383	39043
page.Naut.4096.c	-	15	15	-	4880	15664	10720	41939	123625	1598	2109	926	8036	39084

Tabela 7.13: Número de páginas transmitidas para 16 nós - Técnica de Detecção de Escrita CO-WD

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
SIGS.Tmk.4096	15	15	15	266	785	2239	960	3360	8640	3813	5725	11478	11?	5834
SIGS.JIA.4096	2	2	2	255	750	2175	1024	3136	9216	264	-	780	925	2947
SIGS.Naut.4096.n	15	15	15	18604	87164	329325	1024	4480	9216	42943	76267	124687	9895	85569
SIGS.Naut.4096.c	-	15	15	-	5802	21160	1024	4480	9216	5256	8477	9945	15105	85535

Tabela 7.14: Número de SIGSEGVs para 16 nós - Técnica de Detecção de Escrita CO-WD

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
diffs.Tmk.4096	15	15	15	2639	9914	24655	0	0	0	868	1167	1362	15142	46220
diffs.JIA.4096	15	15	15	0	0	0	0	0	0	343	-	367	6375	348
diffs.Naut.4096.n	15	15	15	0	0	0	0	0	0	300	300	300	3368	1898
diffs.Naut.4096.c	-	15	15	0	0	0	0	0	0	300	300	300	4936	1895

Tabela 7.15: Número de mensagens de *diffs* para 16 nós - Técnica de Detecção de Escrita CO-WD

também se manteve a mesma. Observando-se a tabela 7.13, o número de mensagens relativas a páginas também se mantém. Nota-se também pela tabela 7.14 que o número de SIGs permanece o mesmo. O número de mensagens de *diffs* é constante, conforme pode ser observado na tabela 7.15. Todo este comportamento, quase que idêntico ao tradicional mecanismo de detecção, era esperado já que este *benchmark* praticamente não faz sincronização, onde seria notado o efeito da técnica de detecção CO-WD.

7.3.2 LU (SPLASH-2)

Pelo que se pode observar na figura 5.15 e na tabela 5.4, os *speedups* do Nautilus diminuem para dois casos com a aplicação da técnica de deteção de escrita. Essa diminuição fica em torno de 4,9% para tamanho de matriz de 1792 e 2,7% para tamanho de matriz de 3072. O número de mensagens, como se pode ver na tabela 7.11, reduz-se de 71,3% para tamanho de matriz de 1792 e de 79,7% para tamanho de matriz de 3072. Como pode ser observado na tabela 7.12, o número de kbytes também sofreu reduções de 83,6% (tamanho 1792) e de 82,8% (tamanho 3072), como conseqűência da redução do número de mensagens. Como pode ser constatado na tabela 7.13, o número de páginas que trafegam na rede foi reduzido de 84,7% para matriz de tamanho 1792 e de 82,9%, para matriz de tamanho 3072. O número de SIGs foi reduzido de 93,5% na média (como pode ser constatado na tabela 7.14), conforme esperado. Devido ao maior *overhead* da implementação de se detectar as escritas somente nas páginas que tem cópias do nó *home*, o desempenho do Nautilus é prejudicado

7.3.3 MM

Na figura 5.16 são mostradas as curvas de *speedup* dos vários DSMs, com o Nautilus utilizando a técnica de detecção de escrita CO-WD.

Observando-se a tabela 5.4 e a figura 5.16, os *speedups* do Nautilus aumentaram com a aplicação da detecção de escrita CO-WD. Para matriz de tamanho 1024, o aumento de *speedup* ficou em torno de 1,5%, para matriz de tamanho 1792, um aumento de 5,9% e, finalmente para matriz de 3092, houve um aumento de 0,5%. O número de mensagens, de kbytes, de páginas que trafegam, de SIGs e de *diffs* se manteve-se mais ou menos o mesmo, como pode ser constatado nas tabelas 7.11, 7.12 e 7.13. A manutenção destes parâmetros já era esperada, pois como este programa possui uma única barreira localizada no fim e como a técnica de detecção CO-WD só apresenta resultados após a ocorrência de uma barreira, os efeitos desta técnica não aparecem. Assim, não podem ser diminuídos o número de SIGSEGVs (ou SIGs). A responsabilidade maior da redução de *speedup* deve-se à técnica de detecção de escrita CO-WD, que possui um maior *overhead* da implementação de se detectar as escritas somente nas páginas que tem cópias do nó *home*, prejudicando assim o desempenho do Nautilus.

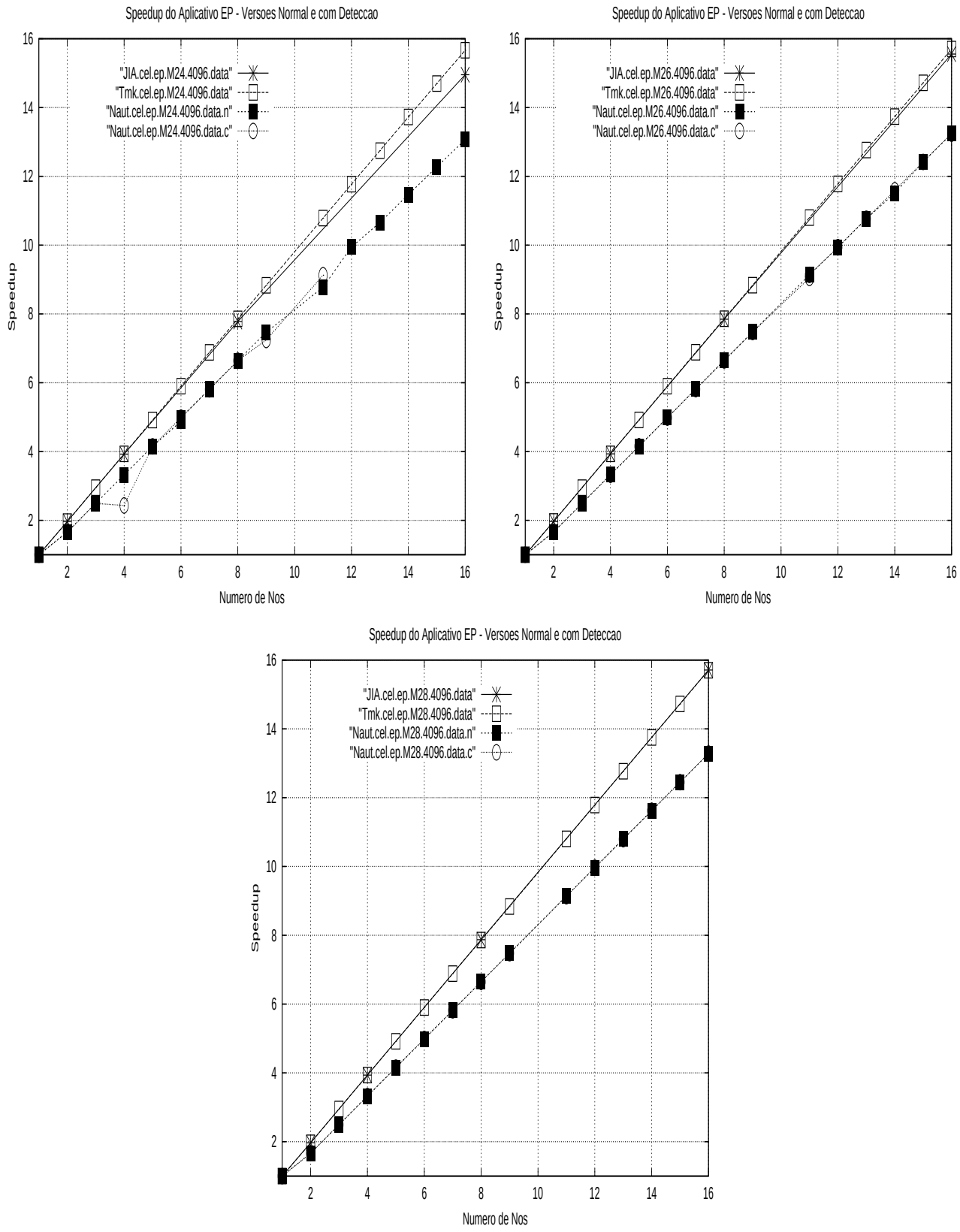

Figura 7.7: *Speedups* do aplicativo EP (NAS) - Técnica de Detecção de Escrita CO-WD

7.3.4 SOR (Rice-University)

Na figura 5.17 são mostradas as curvas de *speedup* do SOR dos vários DSMs, onde o Nautilus utiliza a técnica de detecção de escrita CO-WD.

Pelo que se pode observar da tabela 5.4 e das curvas de *speedup* da figura 5.16, os *speedups* do Nautilus aumentaram de 97,8% para matriz de tamanho 1024, 111,4% para matriz de tamanho 1536 e 138,3%, para matriz de 2048. Como pode ser averiguado na tabela 7.11, o número de mensagens diminui de 9,3% para matriz de 1024, de 19,7% para matriz de 1536 e de 38,1% para matriz de 2048. Pela observação da tabela 7.12, o número de kbytes foi reduzido de 24,5%, 31,2% e 55,7% para os tamanhos de respectivamente 1024, 1536 e 2048. O número de páginas também sofreu reduções de 20,2%, 15,5% e 50,9% (para os tamanhos de 1024, 1536 e 2048) o que pode ser visto na tabela 7.12. E, finalmente pela observação da tabela 7.13, o número de SIGS reduziu-se de 90,0% em média. Devido à grande redução do número de mensagens, redução do número de páginas que trafegam na rede e a grande redução do número de SIGs, isto é pela permanência das páginas no estado *read-write* (não ocorrem escritas nas barreiras nos nós *home*), a conseqüência foi uma melhora brutal dos *speedups*.

7.3.5 Water (SPLASH-2)

Na figura 7.8 e na tabela 5.4, podem ser vistos os *speedups* do aplicativo Water para os vários DSMs, com a técnica de detecção de escrita CO-WD aplicada ao Nautilus.

Como se pode observar pela figura mencionada anteriormente e pela tabela 5.4, os *speedups* do Nautilus diminuem de 1,10% quando a técnica de detecção de escrita CO-WD é aplicada. Através da tabela 7.11, observa-se que o número de mensagens sofre um aumento de 51,4%. Pela observação da tabela 7.12, o número de kbytes sofre uma diminuição de 41,3%. O número de páginas transmitidas sofreu um aumento de 49,3% (tabela 7.13). O número de SIGSEGVs, como se pode observar na tabela 7.14, sofre um aumento de 52,7%. O número de *diffs* sofre um aumento de 46,6% (tabela 7.15). Os resultados esperados da aplicação da técnica de detecção CO-WD, como por exemplo a diminuição do número de SIGSEGVs, não ocorreram. Ao contrário, a técnica fez com que o número de páginas transmitidas aumentasse, causando o aumento do número global de mensagens de páginas, conseqüentemente de mensagens, e de kbytes, por fim, fazendo com que os *speedups* pouco diminuíssem.

7.3.6 Barnes (SPLASH-2)

Na figura 5.18 estão as curvas do aplicativo Barnes para os vários DSMs, onde o Nautilus utiliza a técnica de detecção de escrita CO-WD.

Pelo que pode se observar nessa figura e na tabela 5.4, onde os *speedups* no geral empatam, alguns pontos do gráfico onde o *speedup* melhora com a aplicação da técnica. Como se pode observar na tabela 7.11, o número de mensagens se manteve em 82000. Na tabela 7.12, pode-se

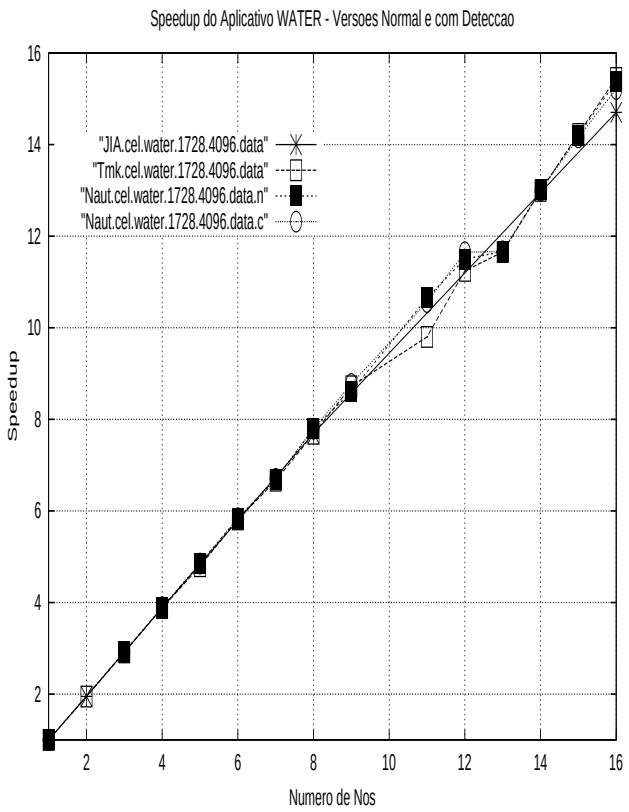

Figura 7.8: *Speedups* do aplicativo Water (SPLASH-2) - Técnica de Detecção de Escrita CO-WD

perceber que o número de kbytes aumentou de 2,2%. O número de mensagens, de mensagens de páginas e de mensagens de *diffs* se mantiveram. O número de kbytes aumentou pelo ligeiro aumento do número de mensagens de páginas, causando o falso compartilhamento, que é o pior caso da conseqüência esperada da aplicação da técnica de detecção CO-WD.

7.4 Técnicas de Agregação de Páginas e Detecção de Escrita CO-WD

Nesta seção será feita a análise detalhada quando aplicadas as técnicas de agregação de páginas e de detecção de escrita CO-WD.

7.4.1 EP (NAS)

A figura 7.9 mostra os *speedups* do aplicativo EP quando submetido aos vários DSMs, incluindo nesta avaliação as técnicas de agregação de páginas e detecção de escrita CO-WD aplicadas conjuntamente ao Nautilus.

Com a aplicação das duas técnicas, como pode ser visto na tabela 5.5, os *speedups* do Nautilus aumentaram de 1,60% para o tamanho de M^{28} (1,91% só a agregação e se mantiveram com só a

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
msgs.Tmk.4096	297	297	297	24590	79339	200052	34845	121309	311494	10427	14923	27109	108976	1171427
msgs.JIA.4096	255	255	255	20764	68146	177562	29034	94142	276847	12168	-	28880	107017	85291
msgs.JIA.8192	255	255	255	14098	46538	153166	13980	46531	138348	7640	9582	19902	85069	45393
msgs.JIA.16384	255	255	255	10786	30410	104674	7020	22636	69003	7640	9582	19902	72156	23967
msgs.JIA.32768	255	255	255	9130	22286	62986	3540	11453	33901	7640	9582	19902	65874	13171
msgs.Naut.4096.n	365	320	320	23092	74730	201558	10872	42084	123947	4968	6400	5032	36200	81988
msgs.Naut.8192.n	332	320	320	16686	50355	165299	5354	20629	61732	3456	4384	5318	43200	44217
msgs.Naut.16384.n	334	318	327	13517	34327	113292	2582	10244	30783	2490	2764	3466	52800	24685
msgs.Naut.32768.n	367	318	320	13477	27238	71717	1271	5062	15382	2560	4200	2456	49600	16317
msgs.Naut.4096.c	-	319	320	-	21393	40898	10752	41971	123657	4508	5142	3114	54819	81645
msgs.Naut.8192.c	320	316	322	8902	15438	31723	5248	20479	61571	3052	3822	4508	51574	44384
msgs.Naut.16384.c	320	320	319	8329	14908	27287	2531	10173	30720	2476	2662	2843	53825	24534
msgs.Naut.32768.c	320	318	320	7797	10448	27287	1250	5099	15344	1995	2203	2452	54289	16071

Tabela 7.16: Número de mensagens para 16 nós - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
kBs.Tmk.4096	91	91	91	17023	46360	107326	69620	243011	624618	16428	24320	48039	31681	131614
kB.JIA.4096	992	992	992	10574	29210	64975	57056	191437	563284	22597	-	65030	137895	162715
kB.JIA.8192	255	255	255	10774	31686	82404	54255	187111	557929	21664	31671	75761	179760	169495
kB.JIA.16384	501	501	501	11779	31446	94414	54008	181217	553867	38546	56234	73922	249666	169255
kB.JIA.32768	992	992	992	14033	33384	94605	53855	182302	542416	72312	105360	137106	393310	168914
kB.Naut.4096.n	56	56	56	29850	121507	374428	43834	168230	495658	8682	11748	8760	59375	167242
kB.Naut.8192.n	107	107	107	39102	140491	593556	42581	164772	493599	9375	13437	20312	62500	172473
kB.Naut.16384.n	227	227	227	55687	198801	764739	40811	163398	492012	9843	15000	21093	140625	166892
kB.Naut.32768.n	452	452	452	96838	235464	813899	39647	162876	491200	10234	19062	19687	234000	209164
kB.Naut.4096.c	-	56	56	-	19879	64431	42880	167756	494500	6556	8082	3881	34839	170948
kB.Naut.8192.c	107	107	107	11029	18532	56468	41721	163574	492314	7076	10195	12921	55538	172023
kB.Naut.16384.c	228	228	228	7842	20976	74452	39980	162255	491006	9208	10911	14328	124328	175229
kB.Naut.32768.c	452	452	452	4820	22177	58727	38666	162155	489989	9454	13952	18344	260062	205225

Tabela 7.17: Número de kbytes transmitida para 16 nós - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
page.Tmk.4096	15	15	15	1820	5460	18790	17325	60542	155599	3963	5912	11780	556	6586
page.JIA.4096	30	30	30	8342	30638	82946	14967	47056	138416	5396	-	13728	32711	44570
page.JIA.8192	30	30	30	5054	19834	70748	7200	23288	69159	3132	4092	9236	21737	23184
page.JIA.16384	30	30	30	3398	11770	46502	3600	11300	34479	3132	4092	5052	15280	11753
page.JIA.32768	30	30	30	2570	7708	25658	1800	5641	16928	3132	4092	4964	12139	5996
page.Naut.4096.n	15	15	15	8359	32072	91669	10840	42058	123915	2003	2497	1885	5383	39043
page.Naut.8192.n	15	15	15	5114	19849	73655	5322	20597	61700	1077	1515	2003	5680	19151
page.Naut.16384.n	15	15	15	3458	11714	47582	2550	10212	30751	570	837	1077	7290	10434
page.Naut.32768.n	15	15	15	3123	8001	26818	1239	5090	15350	1185	1453	572	6115	6250
page.Naut.4096.c	-	15	15	-	4880	15664	10720	41939	123625	1598	2109	926	8036	39084
page.Naut.8192.c	15	15	15	941	2452	6975	5216	20447	61539	870	1255	1598	6535	20257
page.Naut.16384.c	15	15	15	480	1300	4579	2499	10141	30688	570	636	1538	7424	10357
page.Naut.32768.c	15	15	15	303	602	1829	1218	5067	15312	332	371	570	7917	6125

Tabela 7.18: Número de páginas transmitidas para 16 nós - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
SIGS.Tmk.4096	15	15	15	266	785	2239	960	3360	8640	3813	5725	11478	11?	5834
SIGS.JIA.4096	45	45	45	8342	30638	82946	14967	47056	138416	33236	63264	105888	49011	87189
SIGS.JIA.8192	45	45	45	5054	19834	70748	7200	23288	69159	22332	32892	64916	33112	46133
SIGS.JIA.16384	45	45	45	3398	11770	46502	3600	11300	34479	22332	32892	43452	24255	23362
SIGS.JIA.32768	45	45	45	2570	25658	24152	1800	5641	16928	22332	32892	30382	19889	11851
SIGS.Naut.4096.n	15	15	15	18604	87164	329325	1024	4480	9216	42943	76267	124687	9895	85569
SIGS.Naut.8192.n	15	15	15	11796	49169	209133	1024	3136	6144	21737	47715	83903	10459	45804
SIGS.Naut.16384.n	15	15	15	7264	28117	119443	768	2352	4608	11060	24077	42217	12059	23009
SIGS.Naut.32768.n	15	15	15	5736	16440	63152	406	1185	3472	6011	12557	21300	11962	11699
SIGS.Naut.4096.c	-	15	15	-	5802	21160	1024	4480	9216	5256	8477	9945	15105	85535
SIGS.Naut.8192.c	15	15	15	985	2871	8304	1024	3136	6144	2838	4831	7264	12044	45198
SIGS.Naut.16384.c	15	15	15	565	1527	5811	768	2352	4608	1820	2432	7264	13343	22931
SIGS.Naut.32768.c	15	15	15	341	789	2115	406	1185	3472	1138	1402	2332	13324	11672

Tabela 7.19: Número de SIGSEGVs para 16 nós - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

benchmarks	EP	EP	EP	LU	LU	LU	MM	MM	MM	SOR	SOR	SOR	Water	Barnes
parâmetros	M24	M26	M28	1024	1792	3072	1024	1792	3072	1024	1536	2048	1728	16384
diffs.Tmk.4096	15	15	15	2639	9914	24655	0	0	0	868	1167	1362	15142	46220
diffs.JIA.4096	15	15	15	0	0	0	0	0	0	343	354	367	6375	348
diffs.JIA.8192	15	15	15	0	0	0	0	0	0	343	354	370	6375	345
diffs.JIA.16384	15	15	15	0	0	0	0	0	0	343	354	370	6376	343
diffs.JIA.32768	15	15	15	0	0	0	0	0	0	343	354	370	6376	342
diffs.Naut.4096.n	15	15	15	0	0	0	0	0	0	300	300	300	3368	1898
diffs.Naut.8192.n	15	15	15	0	0	0	0	0	0	300	300	300	3691	1775
diffs.Naut.16384.n	15	15	15	0	0	0	0	0	0	300	300	300	4637	1892
diffs.Naut.32768.n	15	15	15	0	0	0	0	0	0	300	300	300	4602	1895
diffs.Naut.4096.c	-	15	15	0	0	0	0	0	0	300	300	300	4936	1895
diffs.Naut.8192.c	15	15	15	0	0	0	0	0	0	300	300	300	4568	1895
diffs.Naut.16384.c	15	15	15	0	0	0	0	0	0	300	300	300	5066	1896
diffs.Naut.32768.c	15	15	15	0	0	0	0	0	0	300	300	300	5081	1897

Tabela 7.20: Número de mensagens de *diffs* para 16 nós - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

detecção de escrita CO-WD), mantendo-se para tamanho de M^{26} e M^{28}. O número de mensagens diminuiu de 12,3% em relação à versão tradicional (4kB e com a detecção tradicional), como pode ser percebido pela tabela 7.16, mantendo-se nos mesmos níveis do que com a aplicação da agregação e da detecção CO-WD. Com relação ao número de kbytes, como mostrado na tabela 7.17, estes aumentaram de 8,1 vezes o valor tradicional (4kB e com a detecção tradicional) , mesmo nível de aumento que somente com a aplicação da agregação. O número de páginas, de SIGs e de *diffs* se mantiveram, como podem ser vistos respectivamente nas tabelas 7.18, 7.19 e 7.20. A justificativa da manutenção dos *speedups* com as técnicas aplicadas conjuntamente é principalmente devido ao comportamento deste aplicativo que transfere pouca quantidade de dados e faz pouca sincronização.

7.4.2 LU (SPLASH-2)

Para o Nautilus, com a aplicação de ambas as técnicas, houve uma redução de *speedup* de 13,54%, 5,81% e 1,31%, para os tamanhos de respectivamente 1024, 1792 e 3072. Pela observação da tabela 7.16, o número de mensagens foi reduzido de até 66,2%, 86,0% e 86,5%, respectivamente para os tamanhos 1024, 1792 e 3072; estas reduções foram maiores que as reduções somente com a aplicação da agregação (67,3%) ou somente com a aplicação da detecção CO-WD (71,3%). Com respeito ao número de kbytes, como pode ser visto na tabela 7.17, em relação à versão tradicional houveram grandes reduções de 82,0% em média, que se compararam em termos percentuais às reduções somente obtidas pela técnica de detecção CO-WD e, quando comparadas à técnica de agregação acompanharam os aumentos percentuais. O número de páginas, como pode ser notado na tabela 7.18, sofreu consideráveis reduções em relação à versão tradicional: 96,4%, 98.1% e 98,0% para tamanho de matriz de 1024, 1792 e 3072. Para o número de mensagens de páginas, comparando-se

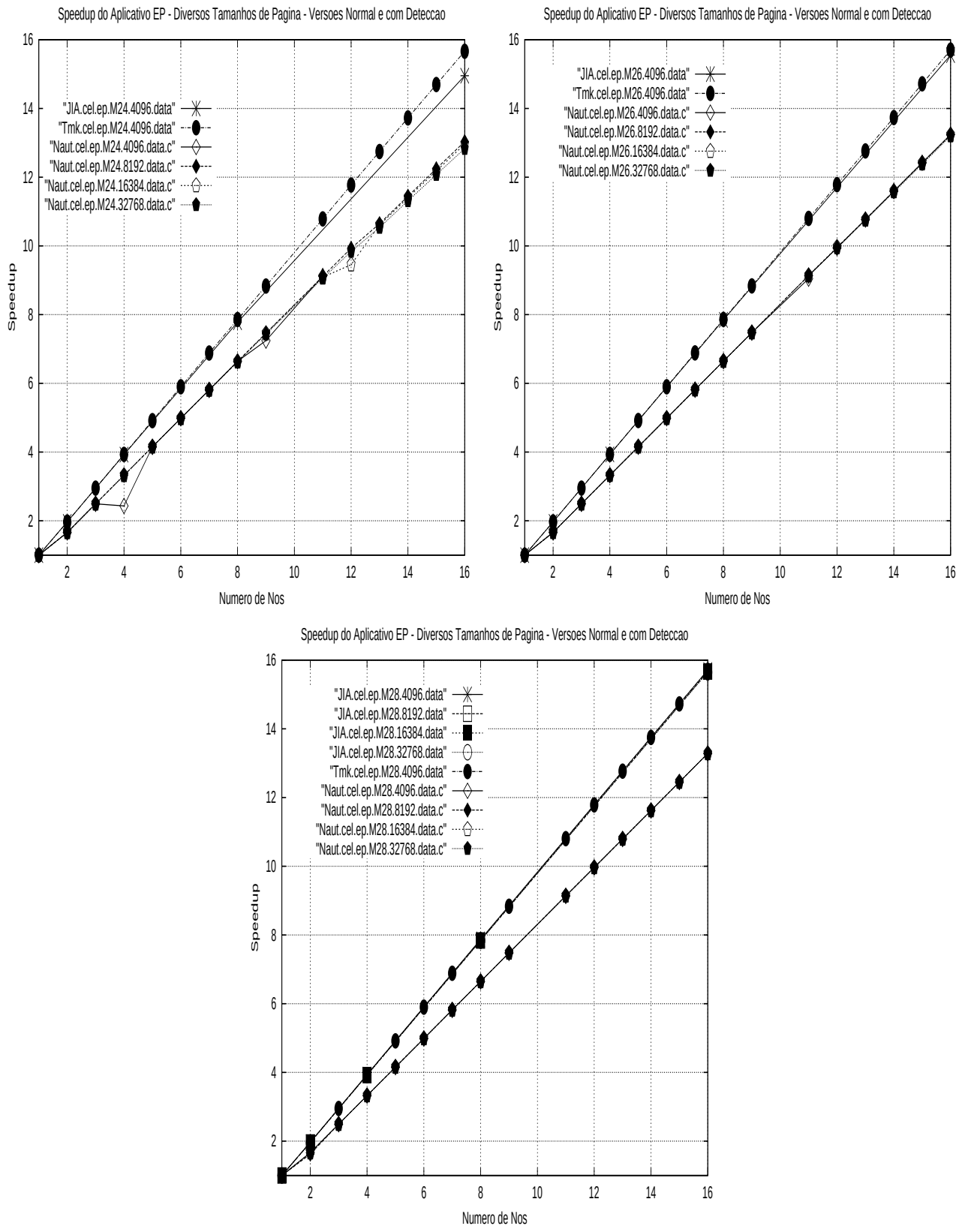

Figura 7.9: *Speedups* do aplicativo EP (NAS) - M^{24}, M^{26}e M^{28} - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

a aplicação conjunta das duas técnicas com a versão que emprega somente a agregação, a redução foi maior (96,4% a 98,1%) do que somente com a aplicação da agregação (71,3%); comparando-se a aplicação conjunta das duas técnicas com a versão somente com detecção CO-WD, também as reduções obtidas foram maiores, isto é, 96,4% a 90,1% contra 82,9% a 84,7%. Como pode ser observado na tabela 7.19, o número de SIGSEGVs foi reduzido de 82,0%, 99,0% e 99,4%, maior que a redução somente com a agregação (69,2% a 81,1%) e maior que somente com a detecção CO-WD aplicada (93,3% em média). O número de mensagens de *diffs* manteve-se mesmo porque quando aplicadas cada uma das técnicas isoladamente, não houveram modificações neste parâmetro.

Concluindo, mesmo com a redução do número de mensagens, devido à redução do número de páginas, do número de kbytes e do número de SIGSEGVs, os *speedups* sofreram redução, que foram predominantemente conseqüência da técnica de detecção CO-WD (como pôde ser visto anteriormente).

7.4.3 MM

Na figura 5.20 estão as curvas de *speedup* para o aplicativo MM submetido aos vários DSMs, avaliando-se a aplicação da agregação e detecção de escrita CO-WD conjuntamente aplicadas ao Nautilus.

Como se pode perceber nesta figura e na tabela 5.5, os *speedups* do Nautilus aumentam de 2,3% e 6,0% para os tamanhos de 1024 e 1792, mantendo-se para o tamanho de 3072. A aplicação conjunta das duas técnicas teve como efeito prático o mesmo resultado da técnica de agregação em relação aos parâmetros que estão sendo observados, já que o programa apresenta somente um ponto de sincronização no final, não permitindo que a parcela de atuação referente à técnica de detecção CO-WD apresente os efeitos desejados. Praticamente a aplicação de ambas as técnicas não apresentou efeitos sobre os parâmetros número de mensagens, de kbytes, de mensagens de páginas, de SIGSEGVs e de mensagens de *diffs* (como pode ser observado nas tabelas 7.16, 7.17, 7.18 e 7.19). Portanto, os aumentos de *speedup* acompanham os mesmos da técnica de agregação. Vale a mesma idéia de considerar o efeito resultante como uma 'soma lógica' dos efeitos das duas técnicas.

7.4.4 SOR (Rice-University)

Na figura 5.21 podem ser vistos os *speedups* do aplicativo SOR submetido aos vários DSMs, onde está sendo avaliada a aplicação de conjunta de ambas as técnicas no Nautilus.

Através da figura mencionada no parágrafo acima e da tabela 5.5, para o Nautllus pode ser concluído que os *speedups* do SOR aumentaram de 53,6% a 102,5%, de 85,5% a 119,8% e de 120,0% a 138,3% (tamanhos de 1025, 1536 e 2048) com a aplicação de ambas as técnicas. Os *speedups*, no geral, foram maiores que os *speedups* com a aplicação somente da agregação e menores do que somente com a aplicação da detecção CO-WD. O número de mensagens, como pode ser observado na tabela 7.16, no geral diminuiu de 59,8%, 65,6% e 51,3%, (tamanhos 1024, 1536 e 2048), ficando

acima das reduções obtidas somente com a agregação (48,5%, 56,8% e 51,2% para os tamanhos de 1024, 1536 e 2048) ou somente com a detecção CO-WD (9,3%, 19,7% e 38,1%). O número de kbytes, como pode ser notado na tabela 7.17, aumentou de 8,9%, 18,8% e 109,0% (tamanhos 1024, 1536 e 2048) em relação à versão tradicional, que são aumentos menores que a versão somente com a agregação (17,9%, 62,3% e 124,7%) e contrários às reduções ocorridas somente com a aplicação da detecção CO-WD (24,5%, 31,2% e 55,7%). Como pode ser observado na tabela 7.18, o número de páginas diminuiu de 83,4%, 85,1% e 69,8%, superiores às reduções obtidas na agregação (71,5% , 66,5% e 69,7%)e da deteção (20,2%, 15,5% e 50,9%). Na tabela 7.19, o número de SIGs sofreu reduções de 97,3%, 98,2% e 98,1%, maiores que as reduções obtidas somente com a aplicação da agregação (86,0%, 83,5% e 82,9%) e maiores que somente com a detecção CO-WD (87,8%, 88,9% e 92,0%). O número de mensagens de *diffs* manteve-se o mesmo, pois a aplicação individual de cada uma das técnicas não apresentou nenhum efeito sobre este parâmetro e portanto, não se esperava que a aplicação conjunta apresentasse algum efeito.

A aplicação conjunta das duas técnicas permitiu agregar o que há de melhor de cada uma delas, dando como resultado um aumento de *speedup*.

7.4.5 Water (SPLASH-2)

Na figura 7.10 estão os *speedups* do aplicativo Water para os vários DSMs, onde estão sendo aplicadas ambas as técnicas conjutamente ao Nautilus.

Pode-se ver pela figura 7.10 e pela tabela 5.5 que os *speedups* do Nautilus apresentaram uma pequena queda de até 3,90% quando ambas as técnicas foram aplicadas. O número de mensagens, mostrado na tabela 7.16, aumentou de até 50,0% em relação ao valor tradicional (4kB e com o mecanismo tradicional de detecção), aumento este maior do que somente com a técnica de agregação (45,9%) e menor do que com a técnica de detecção CO-WD (51,4%). Com relação ao número de kbytes, mostrado na tabela 7.17, este passou a ser 4,4 vezes maior que a versão tradicional, maior do que somente com a agregação (3,4 vezes maior que a tradicional) e contrário à redução obtida somente com a detecção CO-WD (41,3% em relação à tradicional). Com relação ao número de mensagens de páginas que trafegam, como notado na tabela 7.18, houve um aumento de 47,1% em relação à versão tradicional, menor do que o aumento da versão somente com a agregação (35,3%) e menor do que a versão somente com a detecção CO-WD (49,3%). O número de SIGs, como pode ser notado na tabela 7.19, ficou em torno de 34,7% maior que a versão tradicional, maior do que somente com a agregação (21,9% em relação à tradicional) e menor do que somente com a detecção CO-WD (52,7%). Com relação ao número de mensagens de *diffs*, como é exibido na tabela 7.20, houve um aumento de 50,9%, maior do que somente com a agregação (37,7%) e maior do que somente com a detecção CO-WD (46,6%).

Concluindo, pelo aumento do número de mensagens relativas a páginas, como conseqüência direta da aplicação das técnicas de agregação e detecção CO-WD, houve a diminuição de *speedup* do Nautilus.

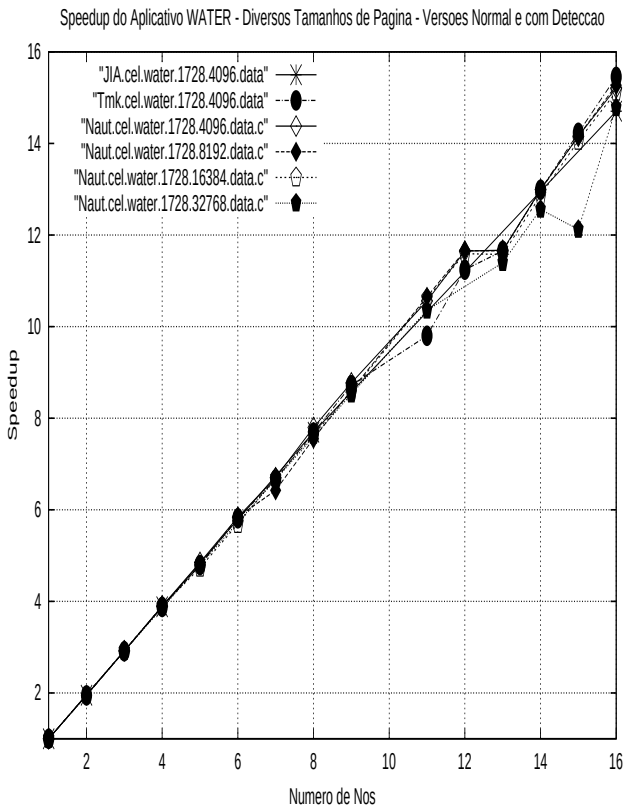

Figura 7.10: *Speedups* do aplicativo Water (SPLASH-2) - 1728 moléculas - Técnicas de Agregação de Página e Detecção de Escrita CO-WD

7.4.6 Barnes (SPLASH-2)

Na figura 5.22 estão os *speedups* do aplicativo Barnes, submetido aos vários DSMs, onde estão sendo avaliadas ambas as técnicas no DSM Nautilus.

Observando-se esta figura e a tabela5.5, os *speedups* tiveram um ligeiro aumento de até 5,0% com a aplicação simultânea das técnicas. O número de mensagens, como pode ser visto na tabela 7.16, sofreu uma redução de 80,4% em relação à versão tradicional, com efeito equivalente à aplicação da agregação. Como a versão com detecção CO-WD não teve efeito neste parâmetro, também a versão com ambas as técnicas conjuntamente aplicadas teve comportamento equivalente a este caso. Como se pode observar na tabela 7.17, com relação ao número de kbytes, houve um aumento médio de 22.7% em relaçao à versão tradicional, isto é, praticamente o mesmo da versão somente com detecção CO-WD e superior ao da versão somente com a agregação (4,2%). Com relação ao número de páginas (tabela 7.18), houve uma diminuição de 84,3% em relação à versão tradicional, ligeiramente superior à versão somente com agregação (83,9%); em contrapartida, a versão somente com detecção CO-WD manteve este parâmetro nos mesmos níveis da versão tradicional. O número de SIGs se reduz de 86,4% em relação à versão tradicional (tabela 7.19), redução esta comparável à soma dos efeitos da redução versão com a agregação (86,3%) e da manutenção da versão com detecção CO-WD. O número de mensagens de *diffs* se manteve praticamente o mesmo (tabela 7.20)com a aplicação de ambas as técnicas, mesmo porque quando aplicadas individualmente não o modificaram.

A conclusão é que os efeitos da aplicação das duas técnicas são a redução do número de mensagens, de mensagens de páginas e de *diffs*, e o aumento do número de kbytes, dando como resultado geral um aumento de *speedup*.

Capítulo 8

Apêndice B - Manual de Usuário do Nautilus

Naut_init()

- cria sockets para a comunicação entre os nós;
- criação dos threads auxiliares para o atendimento das mensagens;
- criação das tabelas de páginas;
- criação de estruturas auxiliares em geral.

Naut_barrier()

Este procedimento permite ao usuário sincronizar todos os seus processos, situados em nós diferentes, por meio de uma barreira.

Após uma barreira o usuário irá notar a manutenção da consistência, isto é, as variáveis compartilhadas vão estar atualizadas.

Naut_lock(<número do semáforo>)

Função que delimita o início de uma seção crítica, utilizada para proteger o acesso a uma variável através de um semáforo binário, escolhido através do parâmetro número do semáforo. O número de semáforos binários disponíveis para sua utilização é 100, porém pode ser modificado pela recompilação da biblioteca do Nautilus.

Naut_unlock(<número do semáforo>)

Esta função delimita o fim da seção crítica, utilizada para proteger o acesso a uma variável através de um semáforo binário.

Após o *unlock*, o usuário irá notar a manutenção da consistência, isto é, as variáveis compartilhadas vão estar atualizadas.

Naut_malloc(<tamanho da memória compartilhada desejada>)

Permite ao usuário alocar suas variáveis compartilhadas. Para isto, o usuário deve especificar o tamanho desejado da mesma.

A sintaxe deste comando é idêntica a do C.

Esta função retorna o tipo void * .

Naut_distribute_data()

Permite a distribuição dos dados compartilhados entre os processadores, pela divisão equalitária dos dados entre os mesmos.

Naut_exit()

Tem as seguintes tarefas:

- finaliza o programa do usuário;
- termina os threads criados;
- desaloca variáveis auxiliares;
- termina com os sockets criados.

Naut_id

Este parâmetro permite ao usuário identificar o número do nó que ele está utilizando. Assim, programas diferentes podem ser executados em nós diferentes.

Naut_nodes

Identifica o número total de nós disponíveis para o usuário.

Exemplo de programa: Matmul

```
#include "bib_Nautilus.h"

    struct timeval tv1, tv2;
    #define TIMER_CLEAR   (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec=0)
    #define TIMER_START   gettimeofday(&tv1, (struct timezone*)0)
    #define TIMER_STOP    gettimeofday(&tv2, (struct timezone*)0)
    #define TIMER_ELAPSED (tv2.tv_sec-tv1.tv_sec+(tv2.tv_usec-tv1.tv_usec)*1.E-6)


    extern pthread_t decoder_thread[Naut_nodes+1],
                     socket_read_thread[Naut_nodes+1];
    pthread_t mult_thread;

    extern char *optarg;
    unsigned int N=0;/*#define N 2048*/

    float **a,**b,**c;


    void initialize()
    {int i,j;

     if (Naut_id==0){
       printf("Initialize matrices!\n");
       for (i=0;i<N;i++)
         for (j=0;j<N;j++){
           a[i][j]=1.0;
           b[i][j]=1.0;
         }
     }
    }

    void parinit()
```

```
{int i,j;
 int start,end;

   printf("Initialize matrices!\n");
   start=(N/Naut_nodes)*Naut_id;
   end=start+(N/Naut_nodes);

   for (i=start;i<end;i++)
     for (j=0;j<N;j++){
       a[i][j]=1.0;
       b[i][j]=1.0;
     }
 }

void printresult()
{int i,j;

 if (Naut_id==0){
   printf("Print matrices!\n");
   for (i=0;i<N;i++)
     for (j=0;j<1;j++)
       /*if (i==0 && ((j % 256) == 0))*/
       printf("c[%d][%d]=%f\n",i,j,c[i][j]);
 }
}

void worker()
{int i,j,k;
 int start, end;
float temp;

 printf("Multiply matrices!\n");

 start=(N/Naut_nodes)*Naut_id;
```

```
 end=start+(N/Naut_nodes);

 printf("start=%d end=%d ", start, end);
 fflush(stdout);
 for (j=0;j<N;j++)
   for (i=start;i<end;i++){
     temp=0.0;
     for (k=0;k<N;k++)
       {
       temp+=a[i][k] * b[j][k]; /* CORRECT! */
       }
       /*temp+=a[i][k]*b[k][j]; /* BAD! */
     c[i][j]=temp;

   }
  printf("arrived ");
  fflush(stdout);
}

void *mult()
{int cc, i,j;
 int pag;

/*
 while ((cc = getopt(argc, argv, "n:")) != -1)
         switch (cc) {
         case 'n':
                 N = atoi(optarg);
                 break;
                }

*/
 if (!N) N=1024;
```

```
 a=(float **)Naut_malloc(N*sizeof(float*));
 b=(float **)Naut_malloc(N*sizeof(float*));
 c=(float **)Naut_malloc(N*sizeof(float*));

 for (i=0;i<N;i++){
   a[i]=(float *)Naut_malloc(N*sizeof(float));
   b[i]=(float *)Naut_malloc(N*sizeof(float));
   c[i]=(float *)Naut_malloc(N*sizeof(float));
 }

 parinit(); /*initialize();*/
 Naut_distribute_data();


 TIMER_CLEAR;
 Naut_barrier();
 TIMER_START;
 worker();

 TIMER_STOP;
 if (Naut_id==0)
   printf("Total time for matrix multiply is == %g seconds\n", TIMER_ELAPSED);
 Naut_barrier();
/*
 printresult();
*/
 if (Naut_id==0)
 printf("Total time for matrix multiply is == %g seconds\n", TIMER_ELAPSED);
 exit(1);

}
```

```
main(argc,argv)
int argc;
char *argv[];
{
  char *Teste;
  char *Teste1;
  char *Teste2;
  char *Mensagem;
  void * retval;

  Naut_init(argc, argv);
  if (pthread_create(&mult_thread, NULL, mult, (void *) NULL ) < 0)
      {
        printf("Error: creating mult_thread\n");
        fflush(stdout);
        exit(1);
      }
 pthread_join(decoder_thread[0], &retval);
 pthread_join(socket_read_thread[0], &retval);
 pthread_join(socket_read_thread[2], &retval);
 pthread_join(mult_thread, &retval);
```